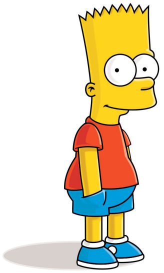
String Matching

String Matching

Problem: Given an alphabet Σ, a *text* $T \in \Sigma^*$ and a *pattern* $P \in \Sigma^*$, find some occurrence/all occurrences of P in T.



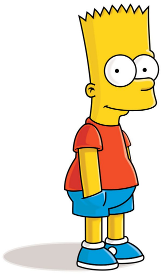
$$\Sigma = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}, \textvisiblespace\}$$

$$T = \texttt{Bart\textvisiblespace played\textvisiblespace darts\textvisiblespace at\textvisiblespace the\textvisiblespace party}$$

$$P = \texttt{art}$$

String Matching

Problem: Given an alphabet Σ, a *text* $T \in \Sigma^*$ and a *pattern* $P \in \Sigma^*$, find some occurrence/all occurrences of P in T.



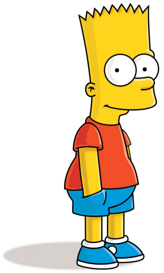
$$\Sigma = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}, \texttt{␣}\}$$

$$T = \texttt{B}\underline{\texttt{art}}\texttt{␣played␣d}\underline{\texttt{art}}\texttt{s␣at␣the␣p}\underline{\texttt{art}}\texttt{y}$$

$$P = \texttt{art}$$

String Matching

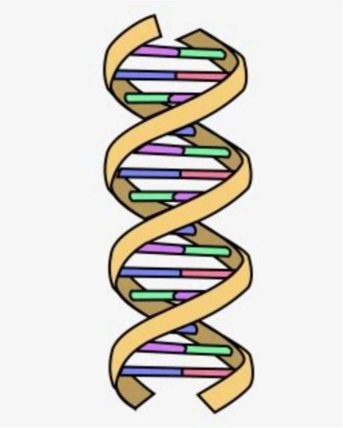
Problem: Given an alphabet Σ, a *text* $T \in \Sigma^*$ and a *pattern* $P \in \Sigma^*$, find some occurrence/all occurrences of P in T.



$$\Sigma = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}, \text{␣}\}$$

$$T = \texttt{B}\underline{\texttt{art}}\text{␣}\texttt{played}\text{␣}\texttt{d}\underline{\texttt{art}}\texttt{s}\text{␣}\texttt{at}\text{␣}\texttt{the}\text{␣}\texttt{p}\underline{\texttt{art}}\texttt{y}$$

$$P = \texttt{art}$$



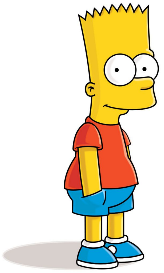
$$\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$$

$$T = \texttt{ACGTGCTTGCAGTGTGCATTACCTGAGTGC}\ldots$$

$$P = \texttt{GTG}$$

String Matching

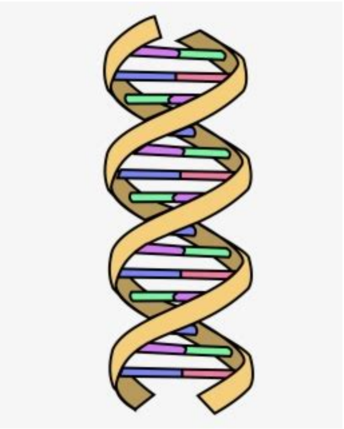
Problem: Given an alphabet Σ, a *text* $T \in \Sigma^*$ and a *pattern* $P \in \Sigma^*$, find some occurrence/all occurrences of P in T.



$$\Sigma = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}, \textvisiblespace\}$$

$$T = \texttt{B}\underline{\texttt{art}}\textvisiblespace\texttt{played}\textvisiblespace\texttt{d}\underline{\texttt{art}}\texttt{s}\textvisiblespace\texttt{at}\textvisiblespace\texttt{the}\textvisiblespace\texttt{p}\underline{\texttt{art}}\texttt{y}$$

$$P = \texttt{art}$$



$$\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$$

$$T = \texttt{AC}\underline{\texttt{GTG}}\texttt{CTTGCA}\underline{\texttt{GT}\underline{\texttt{G}}}\underline{\texttt{TG}}\texttt{CATTACCTGA}\underline{\texttt{GTG}}\texttt{C}\ldots$$

$$P = \texttt{GTG}$$

String Matching

One-shot:

- Both the text and the pattern are **part of the input**
- Algorithm design problem

String Matching

One-shot:

- Both the text and the pattern are **part of the input**
- Algorithm design problem

Repeated:

- The text is **static** and known beforehand (can be preprocessed)
- Patterns are revealed on-demand
- We want to answer each *query* as **quickly** as possible
- Data structure design problem

String Matching

One-shot:

- Both the text and the pattern are **part of the input**
- Algorithm design problem

Repeated:

- The text is **static** and known beforehand (can be preprocessed)
- Patterns are revealed on-demand
- We want to answer each *query* as **quickly** as possible
- Data structure design problem

Tries

Tries (Prounounced as "try")

Data structure to store a dynamic collection of k strings over an alphabet Σ

$$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$$

$$\{\ \mathtt{RAD},\ \mathtt{RADAR},\ \mathtt{RAG},\ \mathtt{RAGE},\ \mathtt{RAGS},\ \mathtt{RATE}\ \}$$

- **Insert**(T): add T to the collection of strings
- **Delete**(T): remove T from the collection of strings
- **Find**(P): return whether P is in the collection

Tries (Prounounced as “try”)

Data structure to store a dynamic collection of k strings over an alphabet Σ

$$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$$

$$\{\ \mathtt{RAD},\quad \mathtt{RADAR},\quad \mathtt{RAG},\ \mathtt{RAGE},\ \mathtt{RAGS},\quad \mathtt{RATE}\ \}$$

- **Insert**(T): add T to the collection of strings
- **Delete**(T): remove T from the collection of strings
- **Find**(P): return whether P is in the collection

Obs: A string comparison requires time $O($string length$)$.
Binary searching requires time $O($max string length $\cdot \log k)$

Tries (Prounounced as "try")

Data structure to store a dynamic collection of k strings over an alphabet Σ

$$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$$

$$\{\ \mathtt{RAD},\ \mathtt{RADAR},\ \mathtt{RAG},\ \mathtt{RAGE},\ \mathtt{RAGS},\ \mathtt{RATE}\ \}$$

- **Insert**(T): add T to the collection of strings
- **Delete**(T): remove T from the collection of strings
- **Find**(P): return whether P is in the collection
- **Count/return** the strings in the collection that start with P

Tries (Prounounced as "try")

Data structure to store a dynamic collection of k strings over an alphabet Σ

$$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$$

$$\{ \mathtt{RAD}, \quad \mathtt{RADAR}, \quad \mathtt{RAG}, \quad \mathtt{RAGE}, \quad \mathtt{RAGS}, \quad \mathtt{RATE} \}$$

- **Insert**(T): add T to the collection of strings
- **Delete**(T): remove T from the collection of strings
- **Find**(P): return whether P is in the collection
- **Count/return** the strings in the collection that start with P
- **Predecessor**(T): return the largest string in the collection that is "not smaller than" T (w.r.t. the lexicopraphic order)

Tries (Prounounced as "try")

Data structure to store a dynamic collection of k strings over an alphabet Σ

$$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$$

$$\{\ \mathtt{RAD},\quad \mathtt{RADAR},\quad \mathtt{RAG},\quad \mathtt{RAGE},\quad \mathtt{RAGS},\quad \mathtt{RATE}\ \}$$

- ~~**Insert**(T): add T to the collection of strings~~
- ~~**Delete**(T): remove T from the collection of strings~~
- **Find**(P): return whether P is in the collection
- **Count/return** the strings in the collection that start with P
- **Predecessor**(T): return the largest string in the collection that is "not smaller than" T (w.r.t. the lexicopraphic order)

We will only focus on the static case

Tries

Pretend that each string ends with a special “end marker” symbol $\$$

RAD   RADAR   RAG   RAGE   RAGS   RATE

Tries

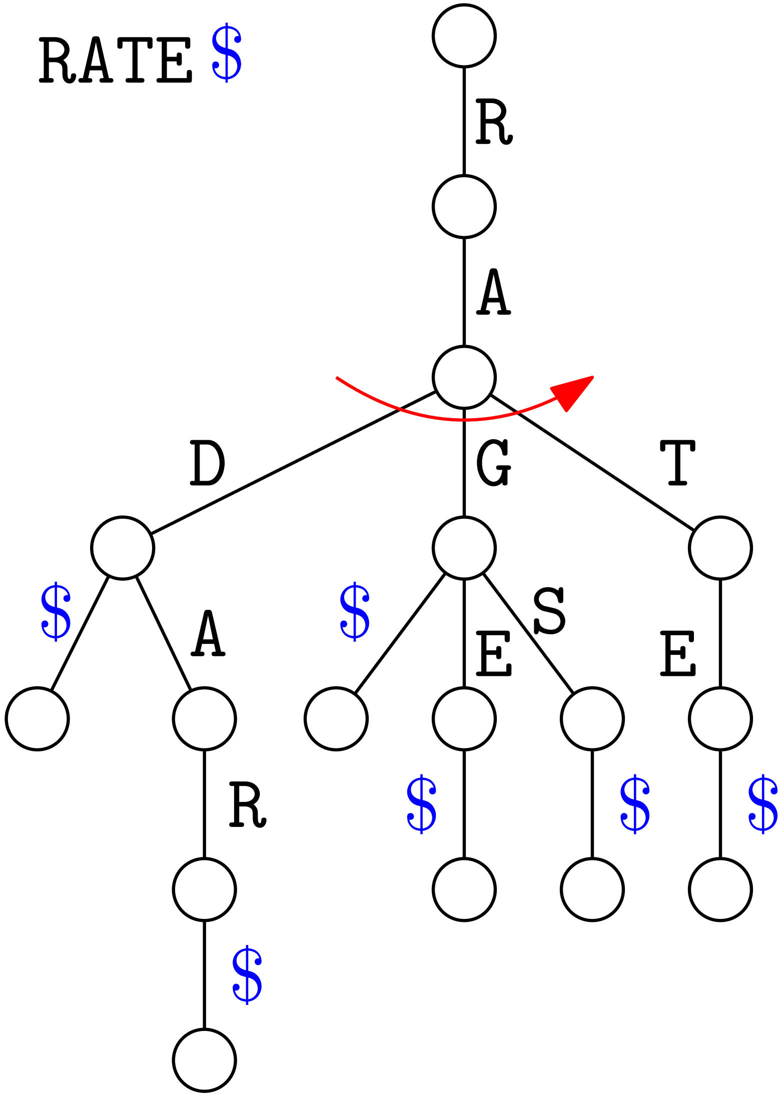
Pretend that each string ends with a special “end marker” symbol $

`RAD$  RADAR$  RAG$  RAGE$  RAGS$  RATE$`

Tries

Pretend that each string ends with a special "end marker" symbol $\$$

RAD$ RADAR$ RAG$ RAGE$ RAGS$ RATE$



Build a tree in which:

- Edges are labelled with a symbol in $\Sigma \cup \{\$\}$ and are sorted

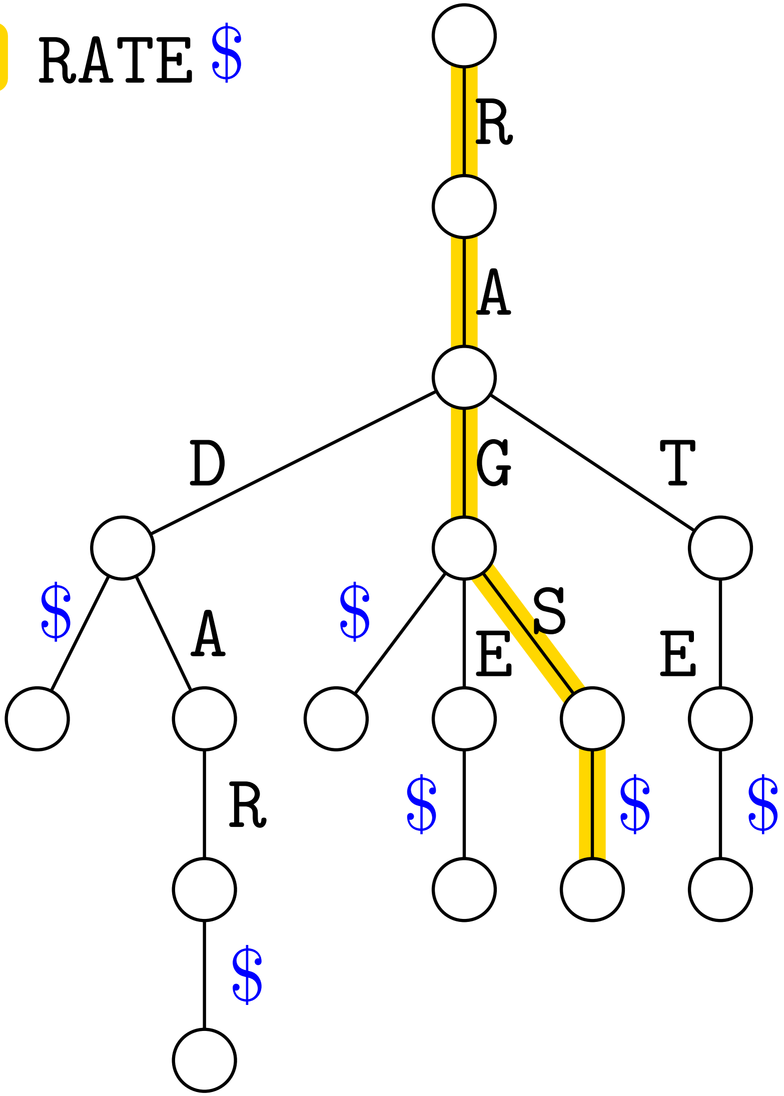
Tries

Pretend that each string ends with a special "end marker" symbol $

RAD$ RADAR$ RAG$ RAGE$ RAGS$ RATE$

Build a tree in which:

- Edges are labelled with a symbol in $\Sigma \cup \{\$\}$ and are sorted
- Each string T_i corresponds to a root-to-leaf path and vice-versa



Tries

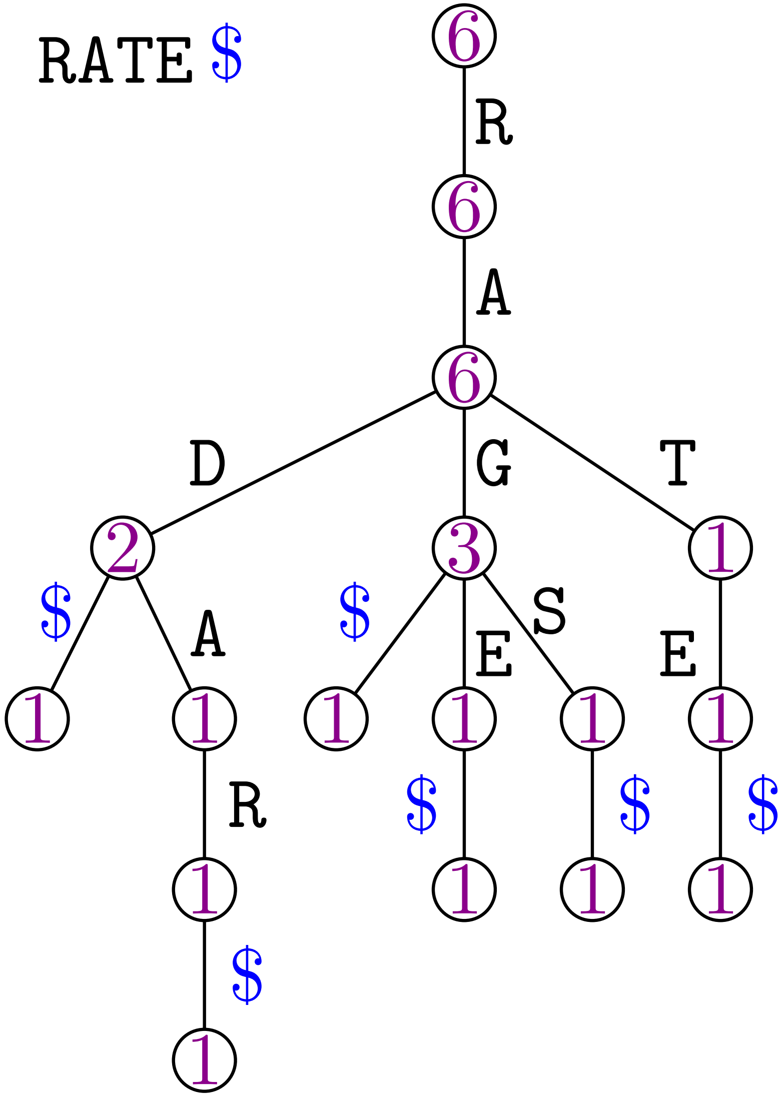
Pretend that each string ends with a special "end marker" symbol $

RAD$ RADAR$ RAG$ RAGE$ RAGS$ RATE$

Build a tree in which:

- Edges are labelled with a symbol in $\Sigma \cup \{\$\}$ and are sorted
- Each string T_i corresponds to a root-to-leaf path and vice-versa
- Satellite data is often useful, e.g.:

— Number of $s in each subtree



Tries

Pretend that each string ends with a special “end marker” symbol $

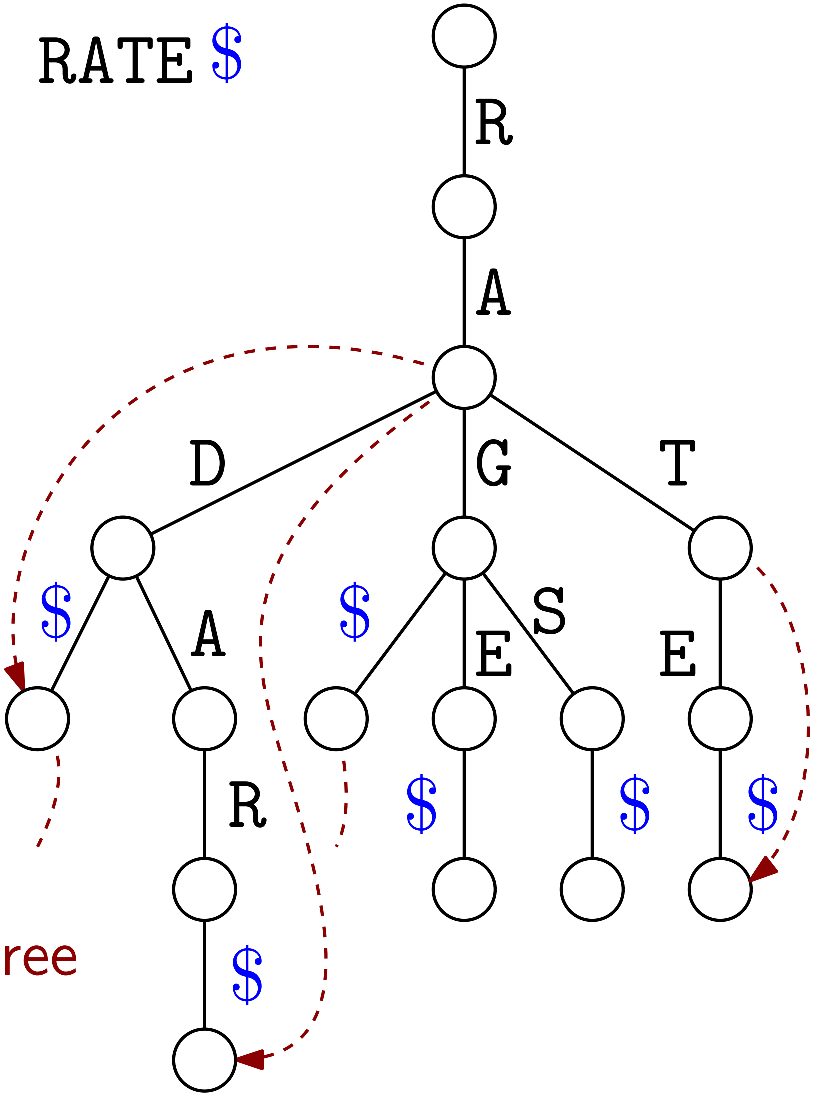
RAD$ RADAR$ RAG$ RAGE$ RAGS$ RATE$

Build a tree in which:

- Edges are labelled with a symbol in $\Sigma \cup \{\$\}$ and are sorted
- Each string T_i corresponds to a root-to-leaf path and vice-versa
- Satellite data is often useful, e.g.:

— Number of $s in each subtree

— Pointers to the first/last leaf in the subtree



Tries

Pretend that each string ends with a special “end marker” symbol $

RAD$ RADAR$ RAG$ RAGE$ RAGS$ RATE$

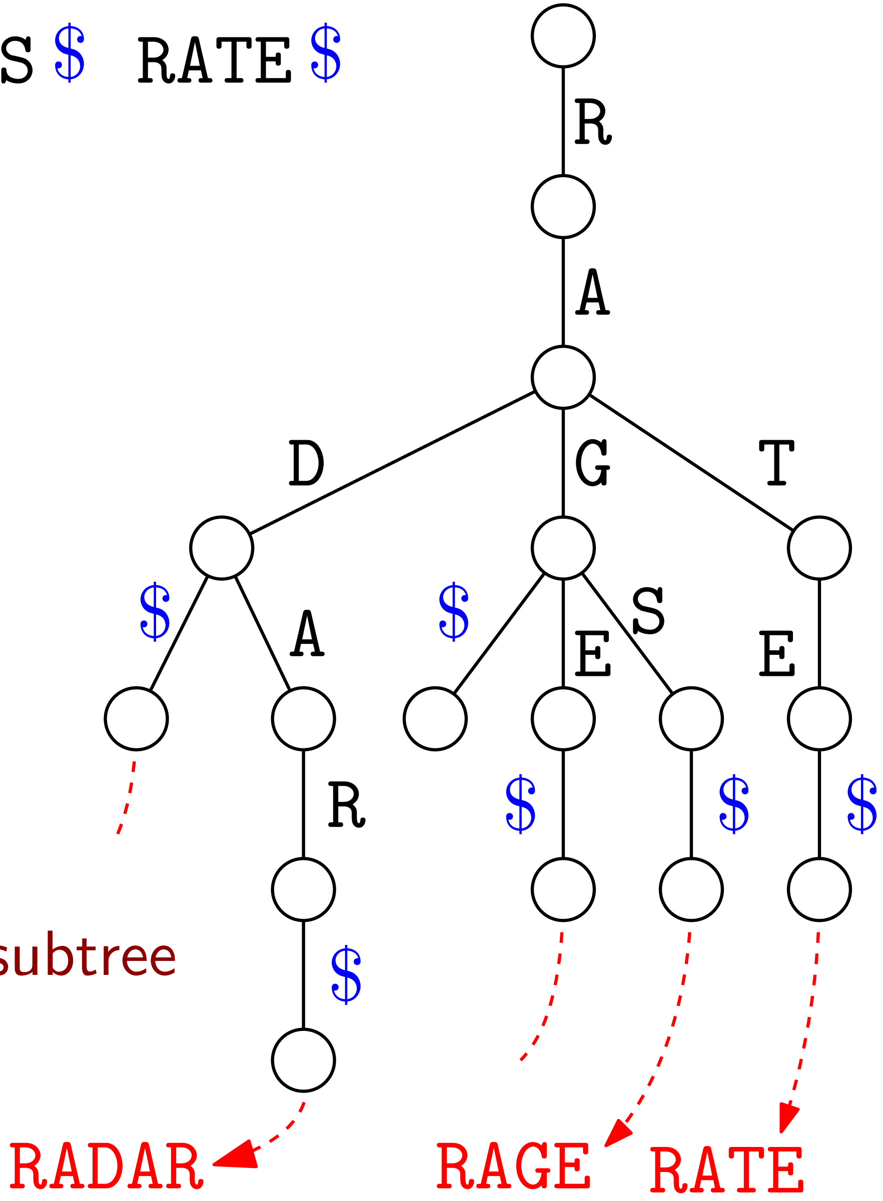
Build a tree in which:

- Edges are labelled with a symbol in $\Sigma \cup \{\$\}$ and are sorted
- Each string T_i corresponds to a root-to-leaf path and vice-versa
- Satellite data is often useful, e.g.:

— Number of $s in each subtree

— Pointers to the first/last leaf in the subtree

— Pointers from leaves to strings



Tries

Pretend that each string ends with a special "end marker" symbol $\$$

RAD$\$$ RADAR$\$$ RAG$\$$ RAGE$\$$ RAGS$\$$ RATE$\$$

Build a tree in which:

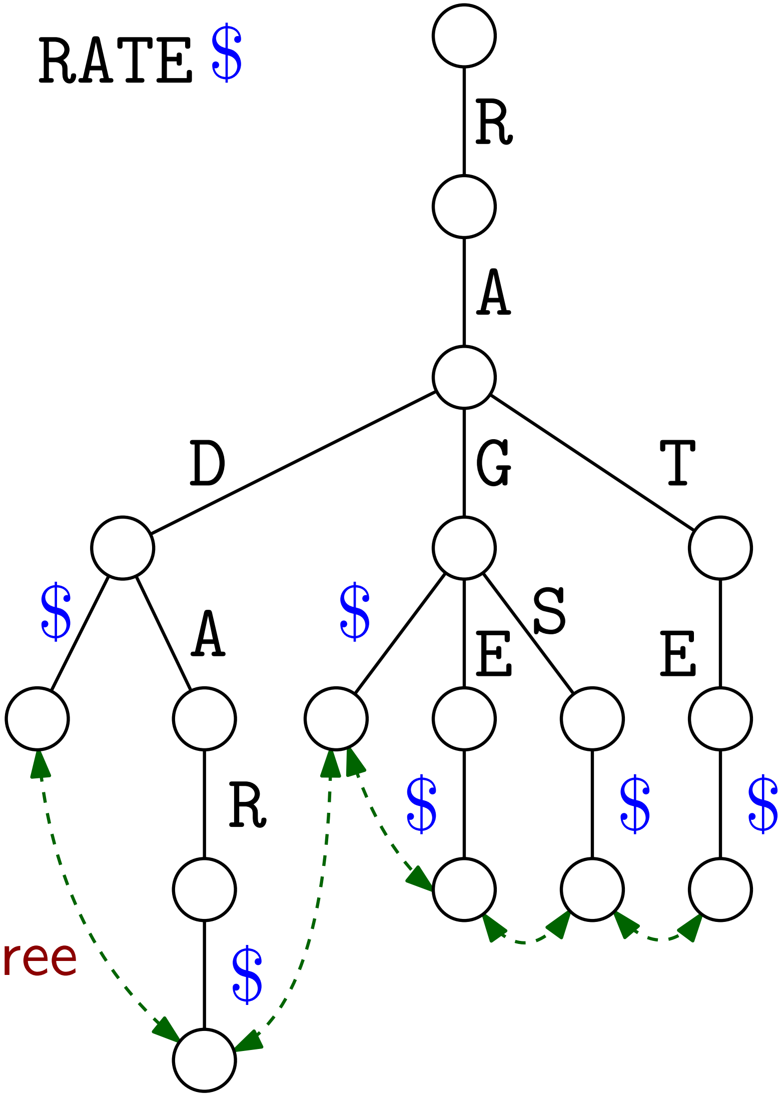
- Edges are labelled with a symbol in $\Sigma \cup \{\$\}$ and are sorted
- Each string T_i corresponds to a root-to-leaf path and vice-versa
- Satellite data is often useful, e.g.:

— Number of $\$$s in each subtree

— Pointers to the first/last leaf in the subtree

— Pointers from leaves to strings

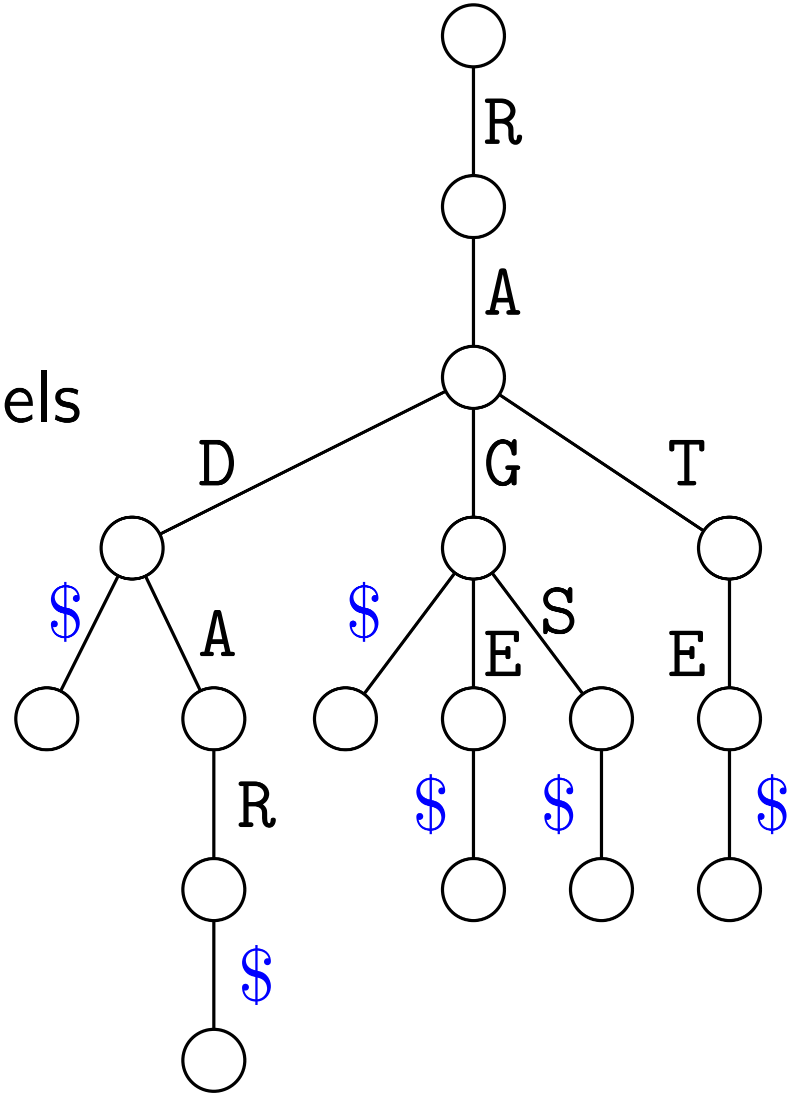
— Leaves arranged in a (doubly) linked list



Tries: Find (Sketch)

Find(P):

- Walk down the tree matching the characters in $P\$$ with the edge labels

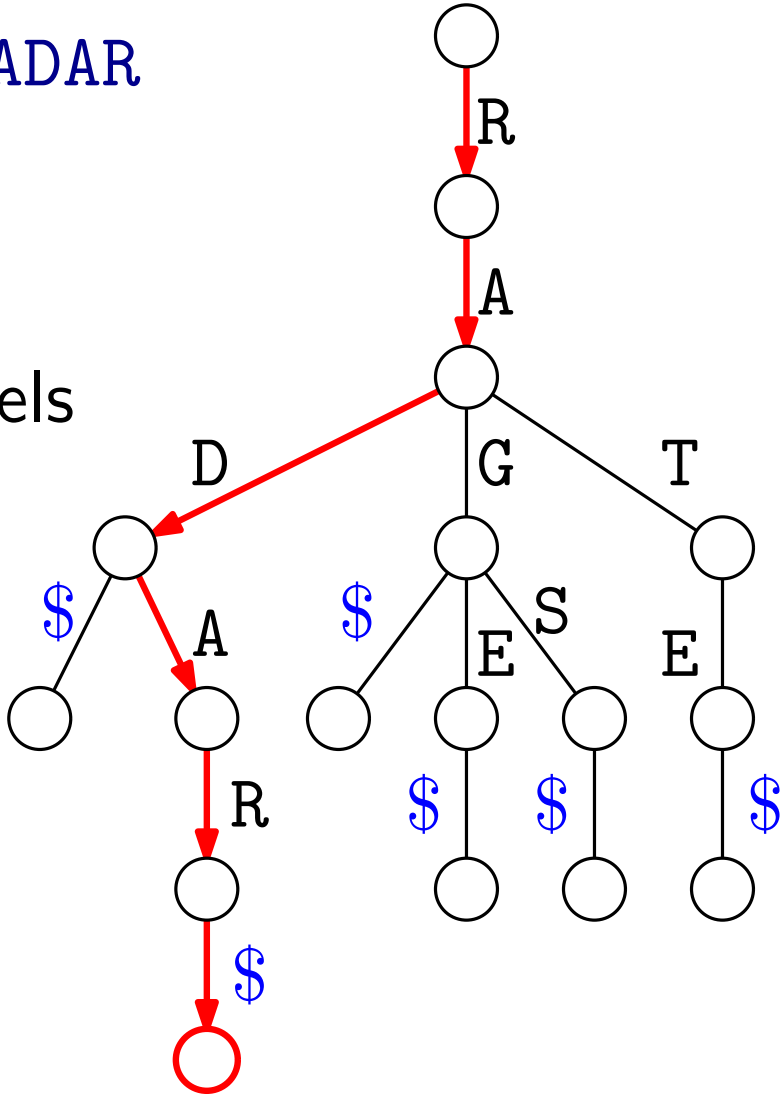


Tries: Find (Sketch)

$P = \texttt{RADAR}$

Find(P):

- Walk down the tree matching the characters in $P\$$ with the edge labels



Tries: Find (Sketch)

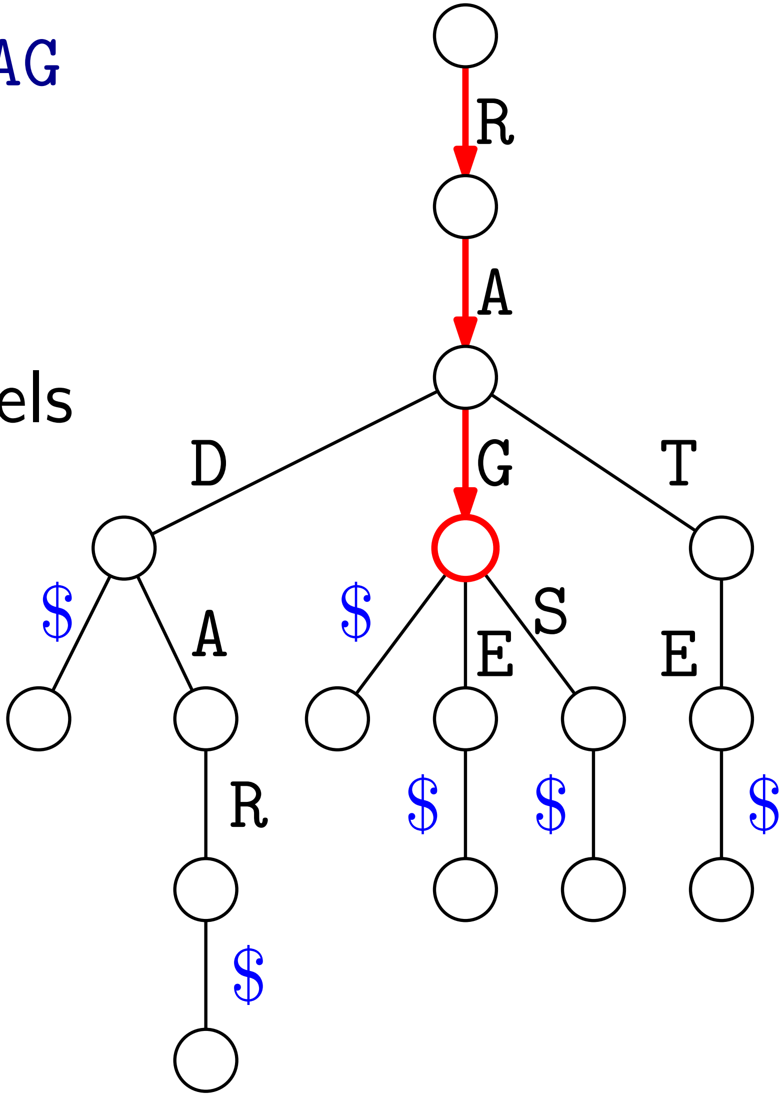
$P = \texttt{RAG}$

Find(P):

- Walk down the tree matching the characters in $P\$$ with the edge labels

To count the number of strings that start with P:

- Find the node corresponding to P

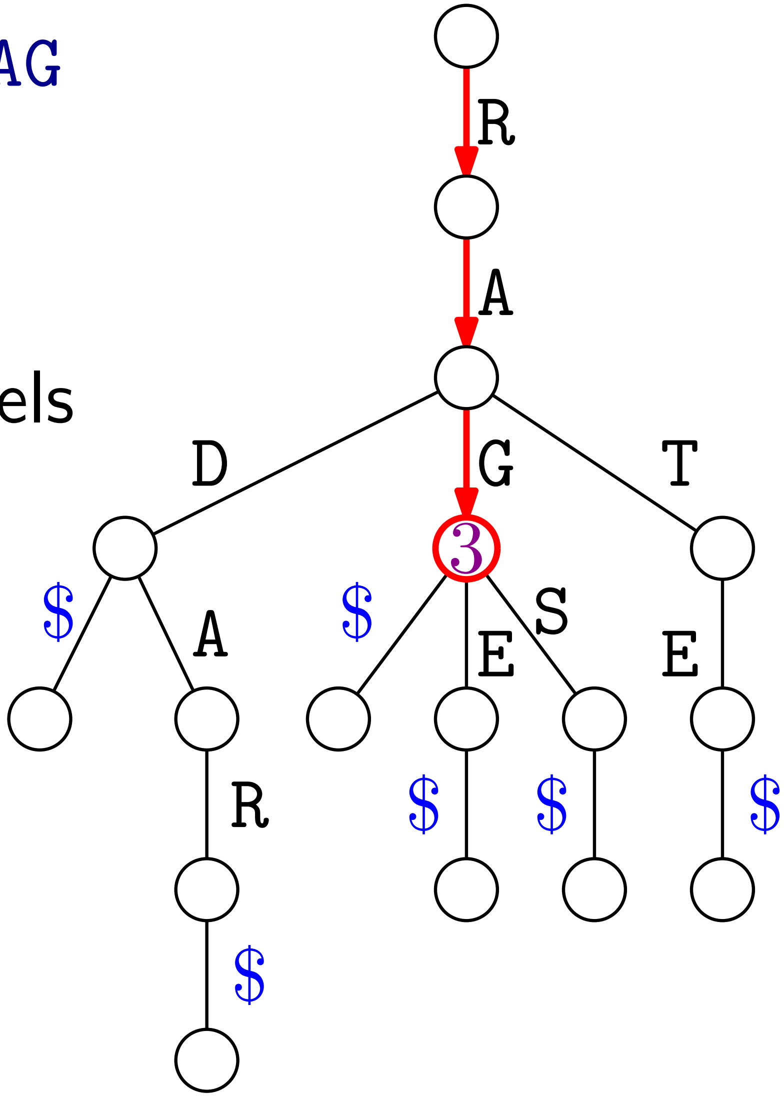


Tries: Find (Sketch)

$P = \texttt{RAG}$

Find(P):

- Walk down the tree matching the characters in $P\$$ with the edge labels

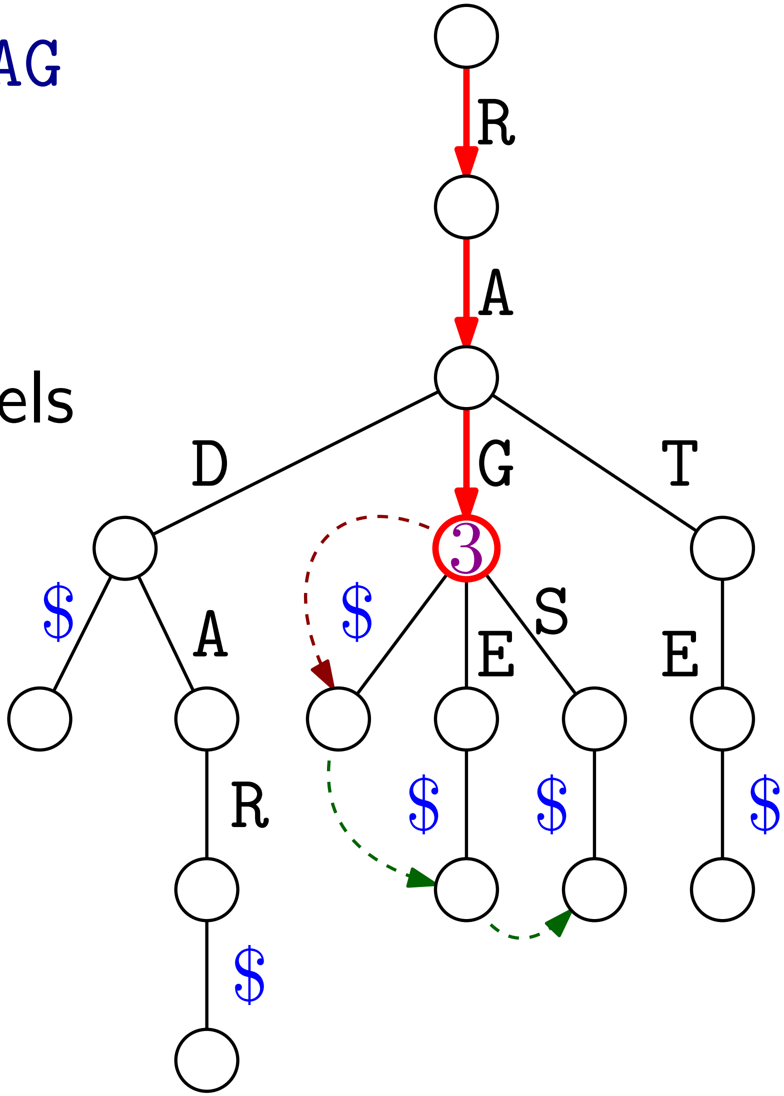


To count the number of strings that start with P:

- Find the node corresponding to P
- Return the number of \$s in the subtree (stored in the node)

Tries: Find (Sketch)

$P = \texttt{RAG}$



Find(P):

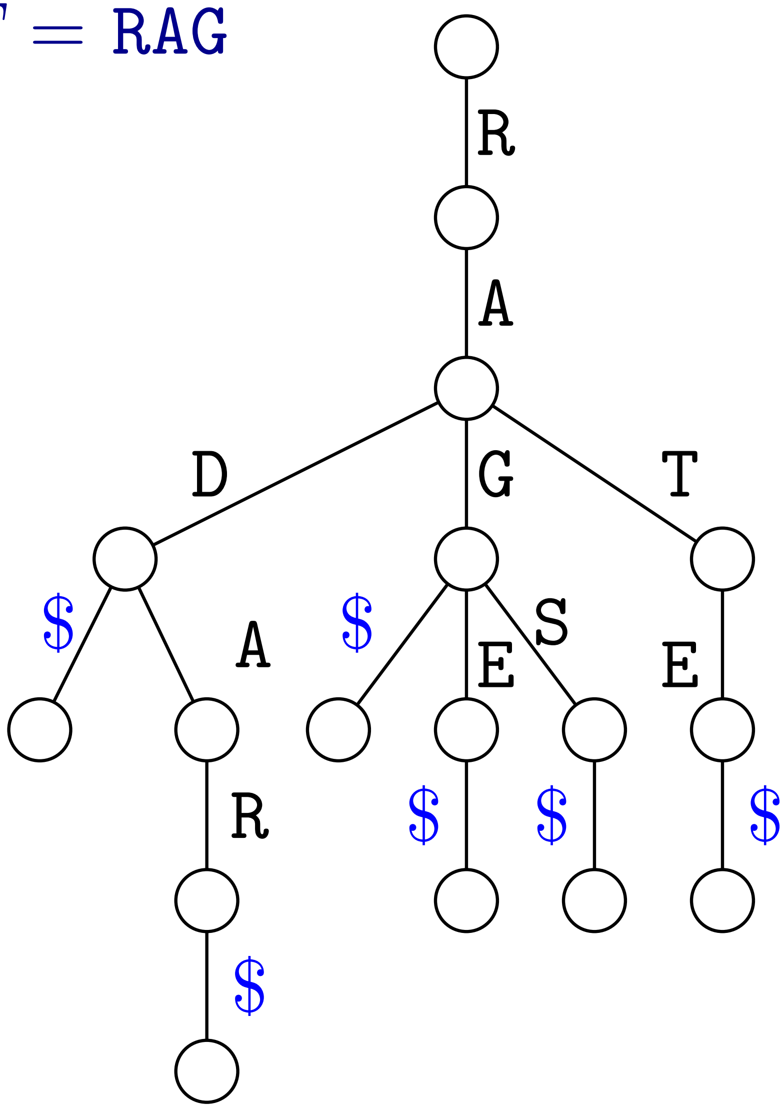
- Walk down the tree matching the characters in $P\$$ with the edge labels

To count the number of strings that start with P:

- Find the node corresponding to P
- Return the number of $s in the subtree (stored in the node)
- The actual matches can be listed in $O(1)$ additional time per match by following pointers

Tries: Predecessor Queries (Sketch)

$T = \texttt{RAG}$

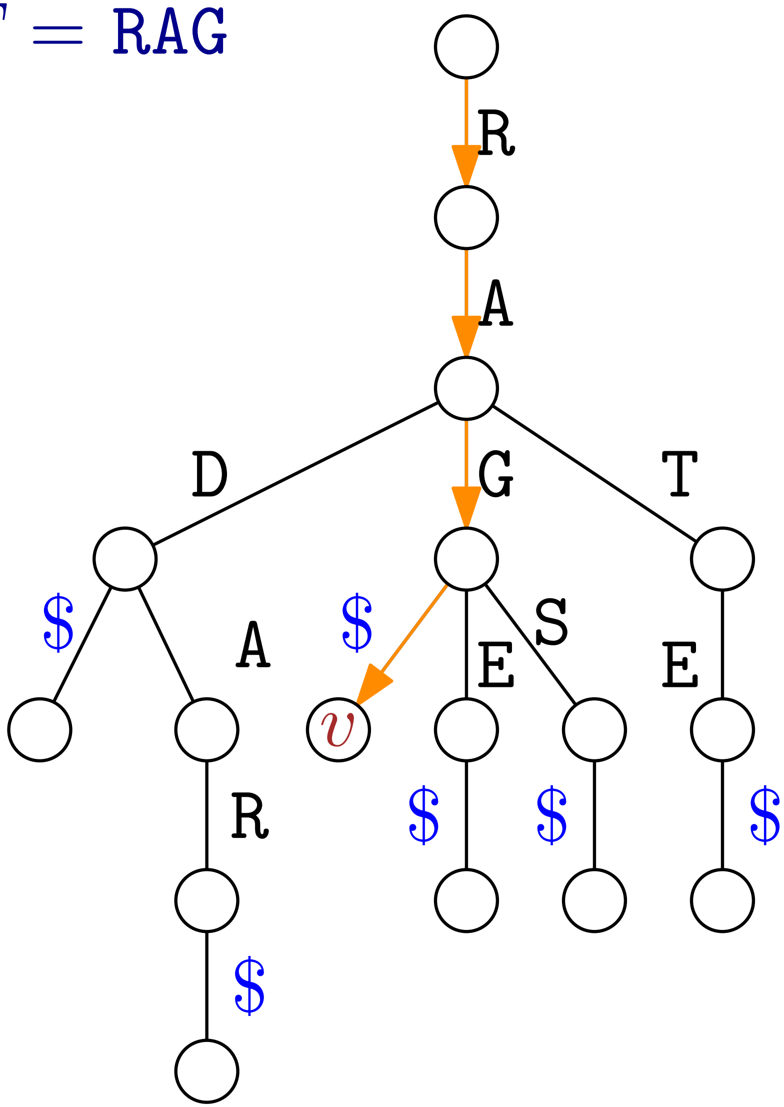


Predecessor(T):

- Walk down a path $\langle v_0, v_1, v_2 \ldots \rangle$ of the tree matching the characters in $T\$$ with the edge labels

Tries: Predecessor Queries (Sketch)

$T = \texttt{RAG}$

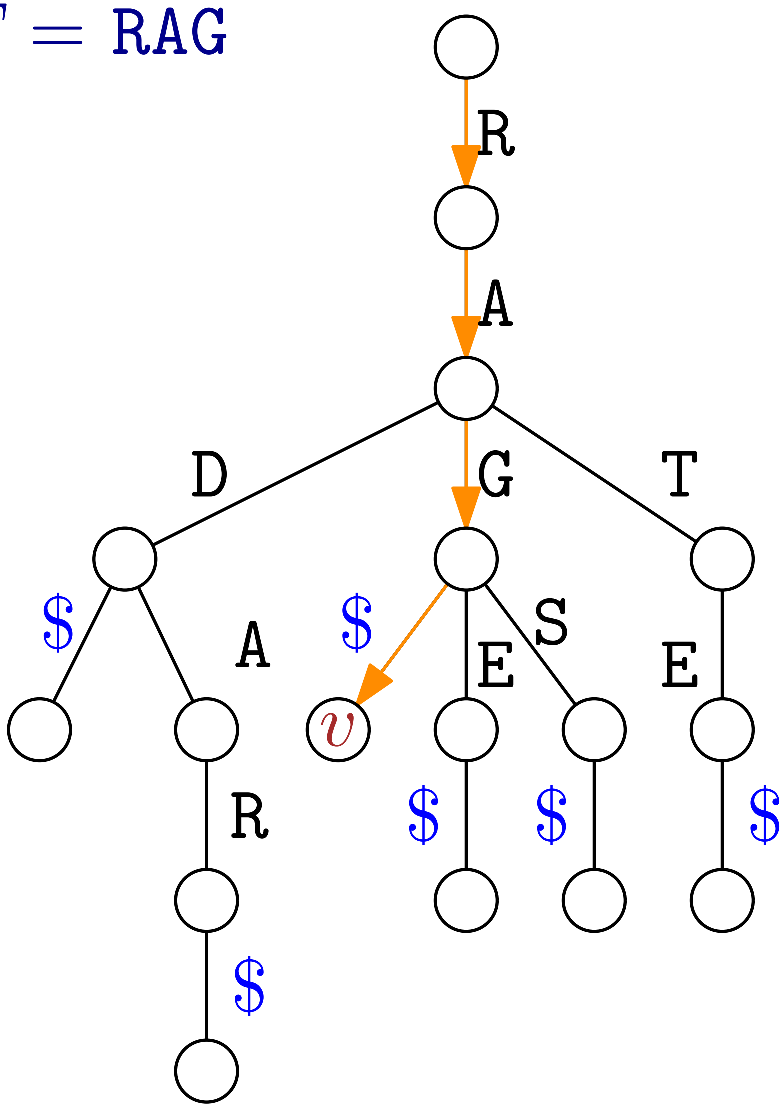


Predecessor(T):

- Walk down a path $\langle v_0, v_1, v_2 \ldots \rangle$ of the tree matching the characters in $T\$$ with the edge labels

Tries: Predecessor Queries (Sketch)

$T = \texttt{RAG}$



Predecessor(T):

- Walk down a path $\langle v_0, v_1, v_2 \ldots \rangle$ of the tree matching the characters in $T\$$ with the edge labels

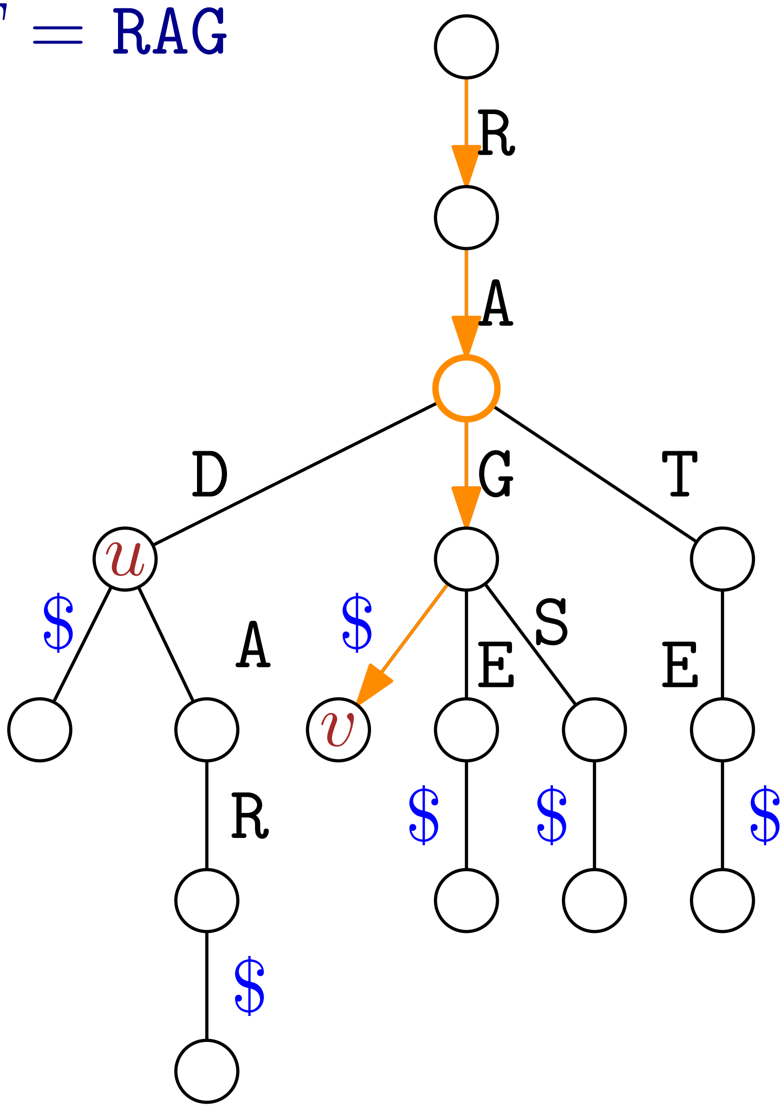
— If $T\$$ is found we are done

— Otherwise, stop at the node v_i matching the longest prefix $T_1 T_2 \ldots T_i$

Tries: Predecessor Queries (Sketch)

$T\$ = T_1 T_2 T_3 \ldots$

$T = \texttt{RAG}$



Predecessor(T):

- Walk down a path $\langle v_0, v_1, v_2 \ldots \rangle$ of the tree matching the characters in $T\$$ with the edge labels

— If $T\$$ is found we are done

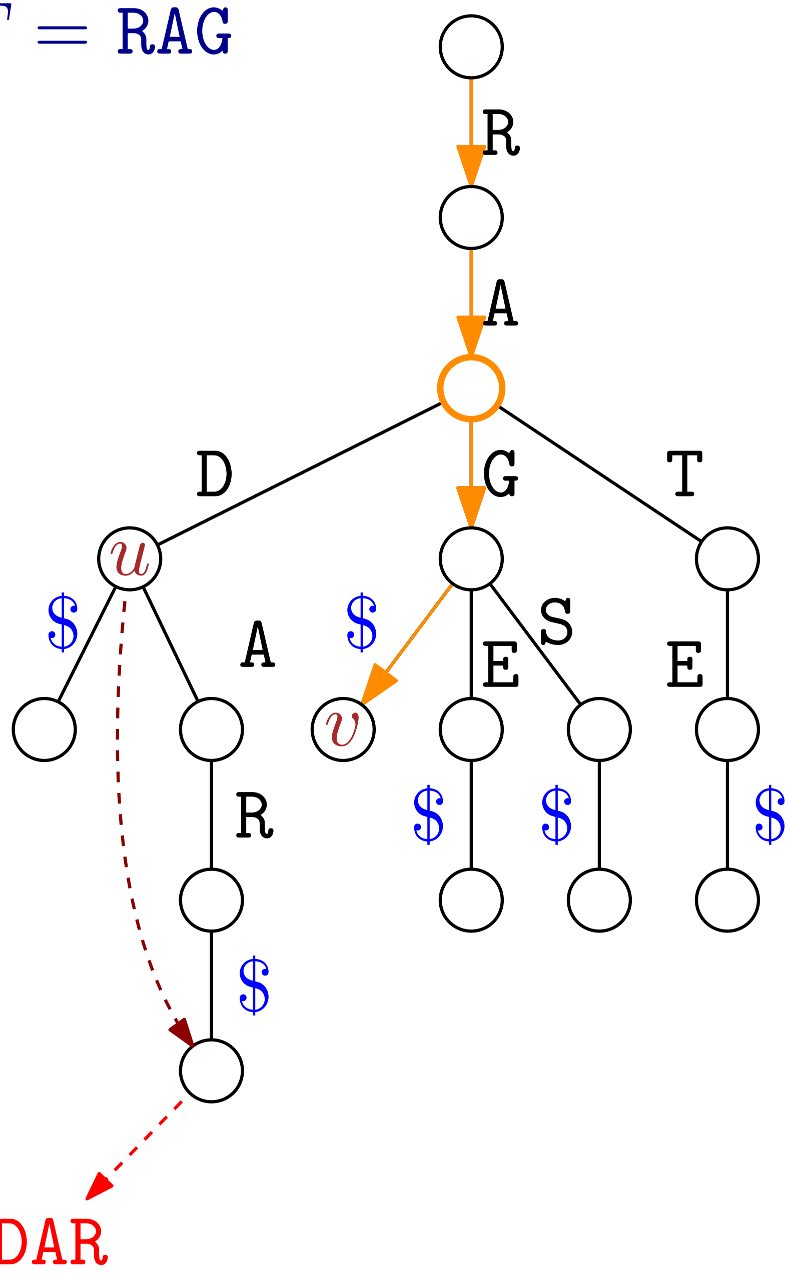
— Otherwise, stop at the node v_i matching the longest prefix $T_1 T_2 \ldots T_i$

- Find the deepest ancestor of v_j of v_i (possibly v_i itself) such that T_j has a strict predecessor u w.r.t. v_j.

Tries: Predecessor Queries (Sketch)

$T\$ = T_1 T_2 T_3 \dots$

$T = \texttt{RAG}$



Predecessor(T):

- Walk down a path $\langle v_0, v_1, v_2 \dots \rangle$ of the tree matching the characters in $T\$$ with the edge labels

— If $T\$$ is found we are done

— Otherwise, stop at the node v_i matching the longest prefix $T_1 T_2 \dots T_i$

- Find the deepest ancestor of v_j of v_i (possibly v_i itself) such that T_j has a strict predecessor u w.r.t. v_j.
- Follow the pointers from u to the maximum string in its subtree

Tries: Predecessor Queries (Sketch)

$T\$ = T_1 T_2 T_3 \ldots$ $T = \texttt{RAG}$

The strict predecessor of $\sigma \in \Sigma$ w.r.t. a node v, if it exists, is the child u of v such that (v, u) has the largest label that is smaller than σ

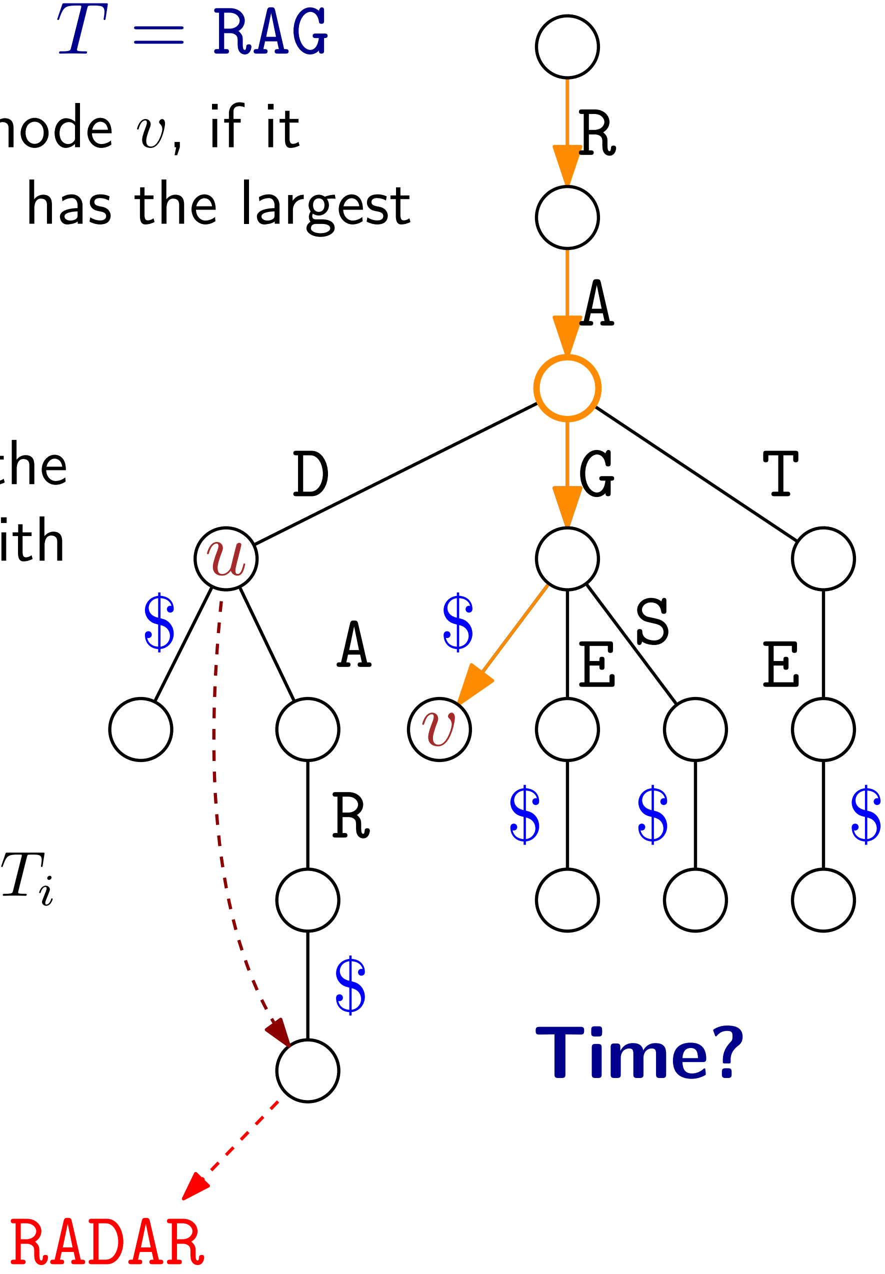
Predecessor(T):

- Walk down a path $\langle v_0, v_1, v_2 \ldots \rangle$ of the tree matching the characters in $T\$$ with the edge labels

— If $T\$$ is found we are done

— Otherwise, stop at the node v_i matching the longest prefix $T_1 T_2 \ldots T_i$

- Find the deepest ancestor of v_j of v_i (possibly v_i itself) such that T_j has a strict predecessor u w.r.t. v_j.
- Follow the pointers from u to the maximum string in its subtree



Time?

Tries: Predecessor Queries (Sketch)

$T\$ = T_1 T_2 T_3 \ldots$

$T = \mathtt{RAG}$

The strict predecessor of $\sigma \in \Sigma$ w.r.t. a node v, if it exists, is the child u of v such that (v, u) has the largest label that is smaller than σ

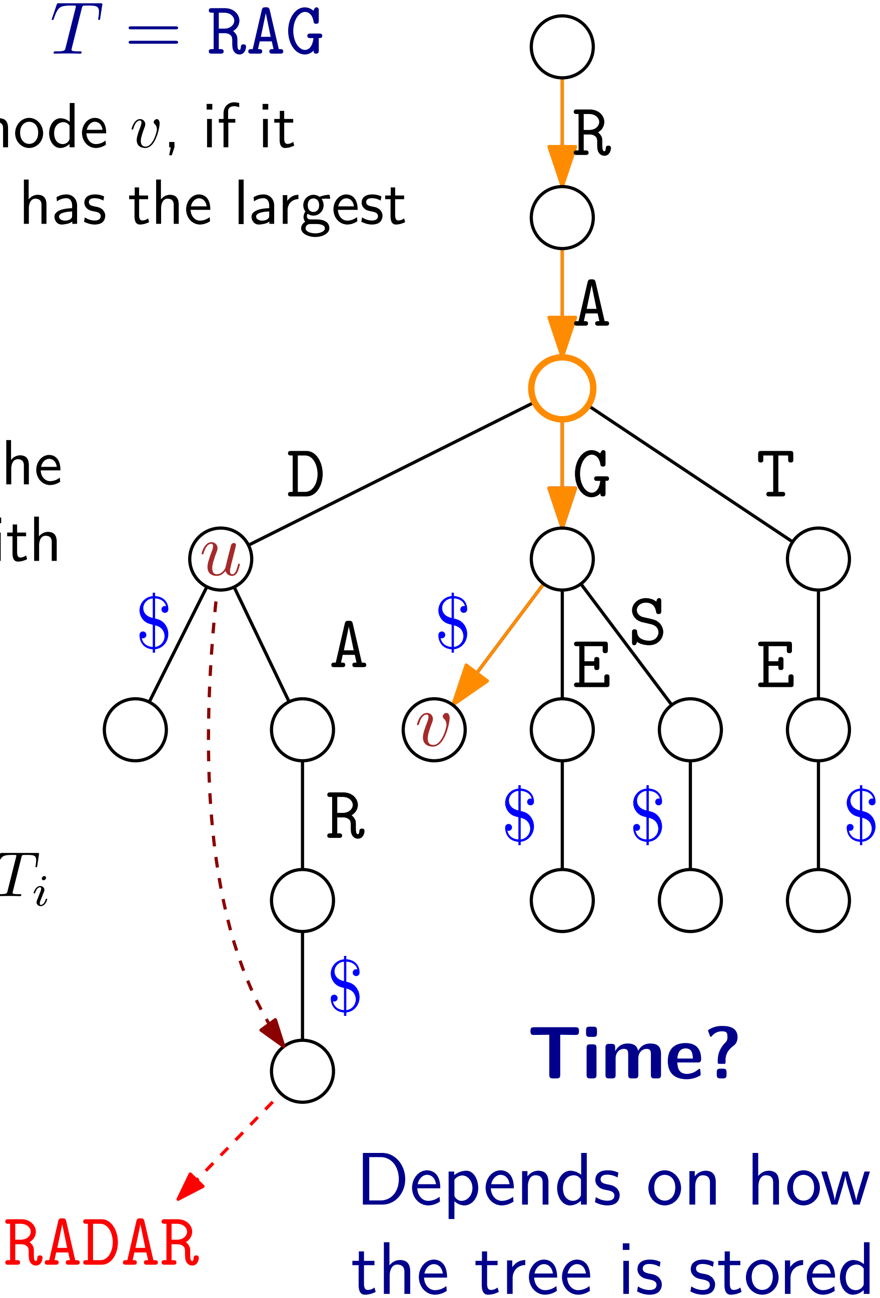
Predecessor(T):

- Walk down a path $\langle v_0, v_1, v_2 \ldots \rangle$ of the tree matching the characters in $T\$$ with the edge labels

— If $T\$$ is found we are done

— Otherwise, stop at the node v_i matching the longest prefix $T_1 T_2 \ldots T_i$

- Find the deepest ancestor of v_j of v_i (possibly v_i itself) such that T_j has a strict predecessor u w.r.t. v_j.
- Follow the pointers from u to the maximum string in its subtree

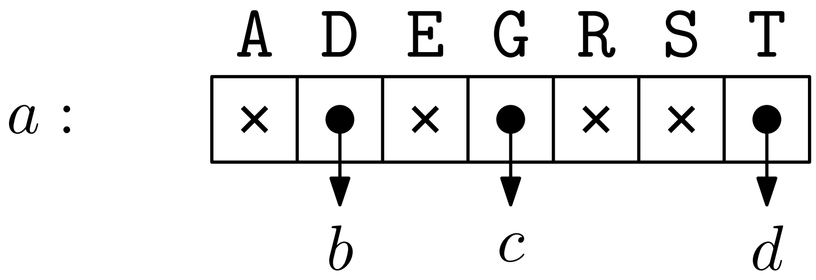


Time?

Depends on how the tree is stored

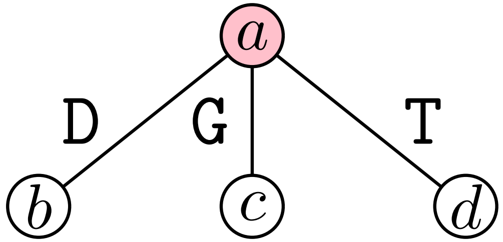
Representing Tries

Array (dense)



$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\texttt{A}, \texttt{D}, \texttt{E}, \texttt{G}, \texttt{R}, \texttt{S}, \texttt{T}\}$

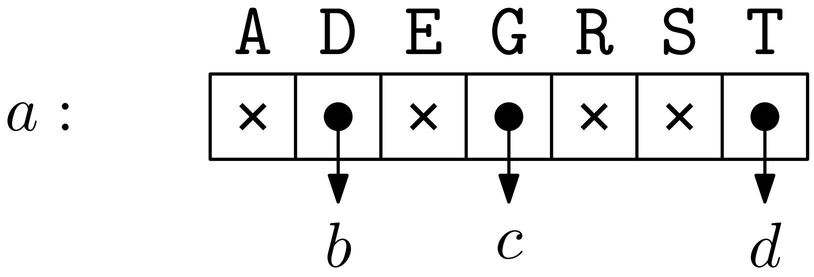


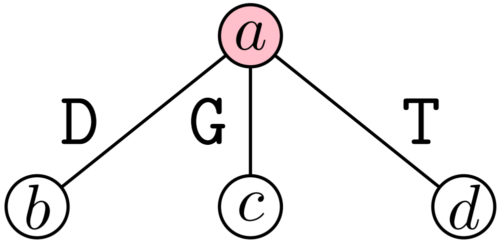
Representing Tries

Array (dense)

$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$





Space: $O(|\Sigma|)$

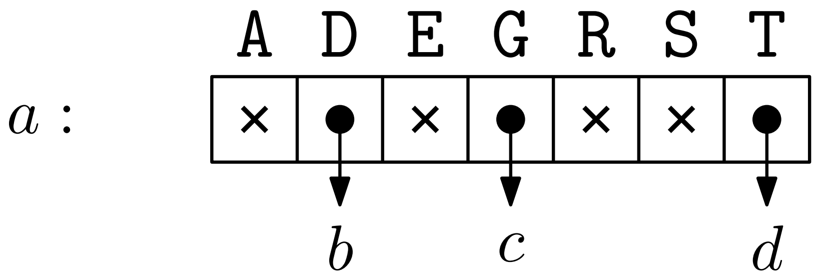
Time to find a symbol's edge: $O(1)$

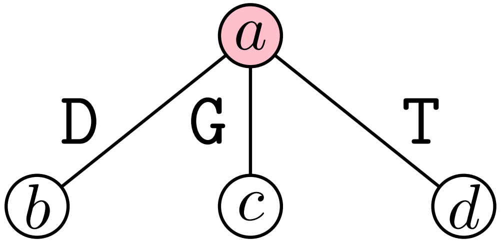
Representing Tries

Array (dense)

$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$





Space: $O(|\Sigma|)$

Time to find a symbol's edge: $O(1)$

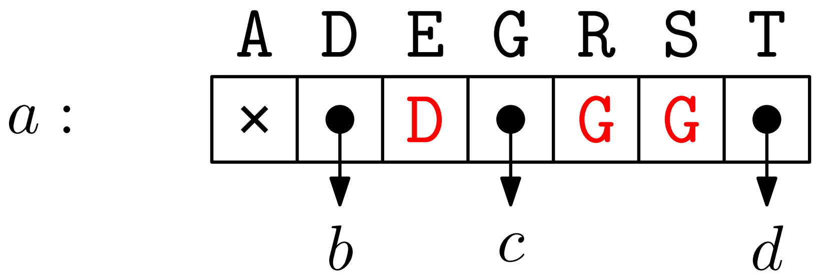
Time to find predecessor: $O(|\Sigma|)$

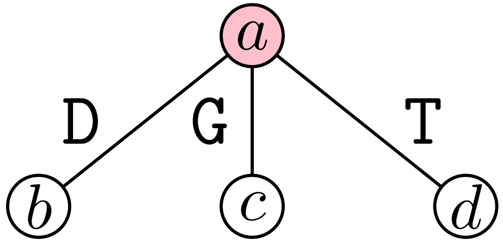
Representing Tries

Array (dense)

$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\texttt{A}, \texttt{D}, \texttt{E}, \texttt{G}, \texttt{R}, \texttt{S}, \texttt{T}\}$





Space: $O(|\Sigma|)$

Time to find a symbol's edge: $O(1)$

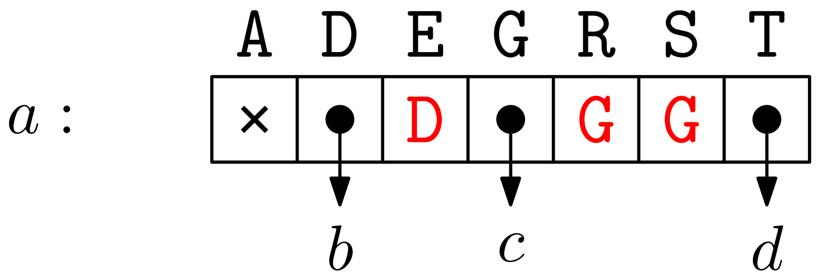
Time to find predecessor: $O(|\Sigma|)$

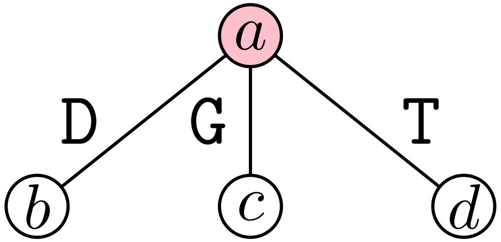
Representing Tries

Array (dense)

$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$





Space: $O(|\Sigma|)$

Time to find a symbol's edge: $O(1)$

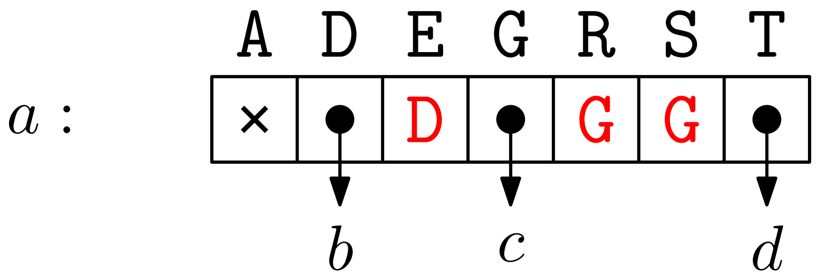
Time to find predecessor: ~~$O(|\Sigma|)$~~ $O(1)$

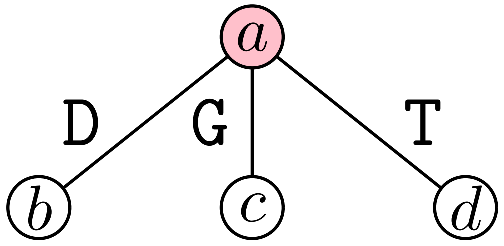
Representing Tries

Array (dense)

$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$





Space: $O(|\Sigma|)$

Time to find a symbol's edge: $O(1)$

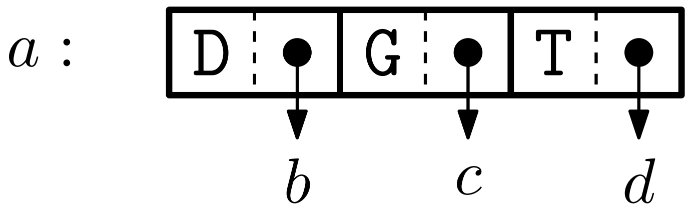
Time to find predecessor: ~~$O(|\Sigma|)$~~ $O(1)$

Overall space: $O(|\Sigma| \cdot n)$

Overall time: $O(|P|)$

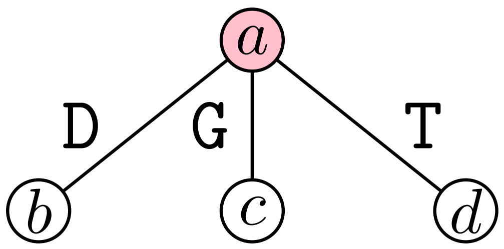
Representing Tries

Array (sparse)



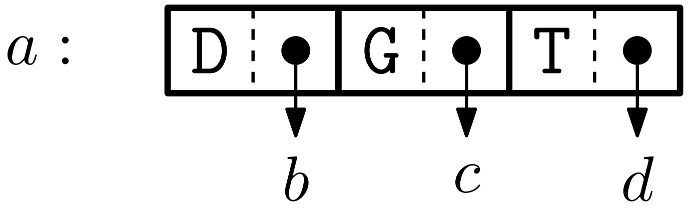
$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\texttt{A}, \texttt{D}, \texttt{E}, \texttt{G}, \texttt{R}, \texttt{S}, \texttt{T}\}$

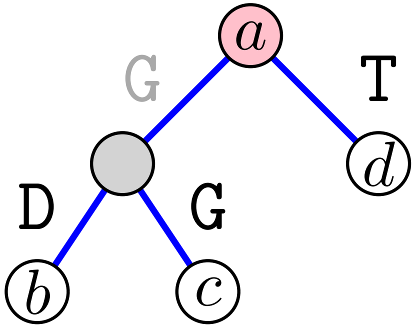


Representing Tries

Array (sparse)

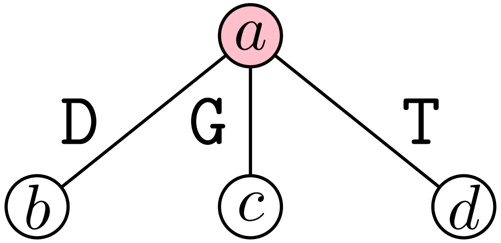


Balanced Binary Search Tree



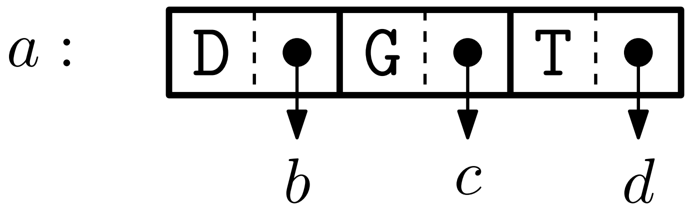
$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$

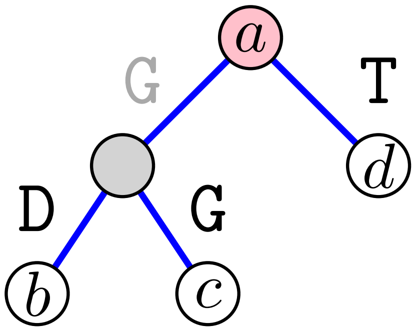


Representing Tries

Array (sparse)



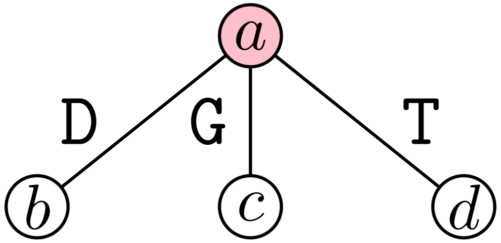
Balanced Binary Search Tree



Space: $O(\#\text{children})$

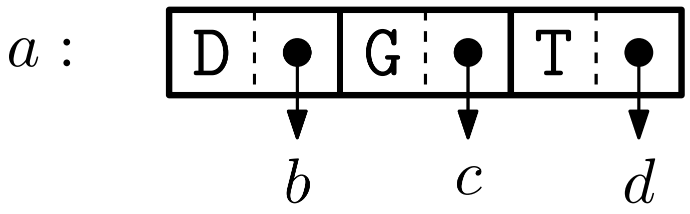
$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$



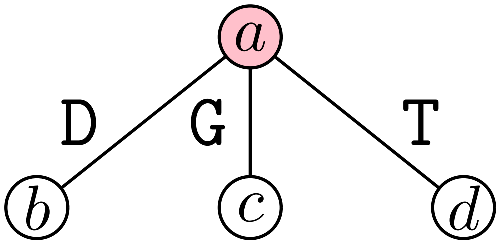
Representing Tries

Array (sparse)

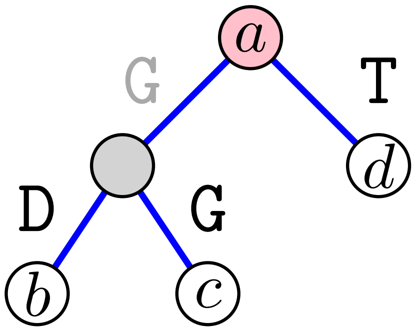


$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

$\Sigma = \{\texttt{A}, \texttt{D}, \texttt{E}, \texttt{G}, \texttt{R}, \texttt{S}, \texttt{T}\}$



Balanced Binary Search Tree



Space: $O(\#\text{children})$

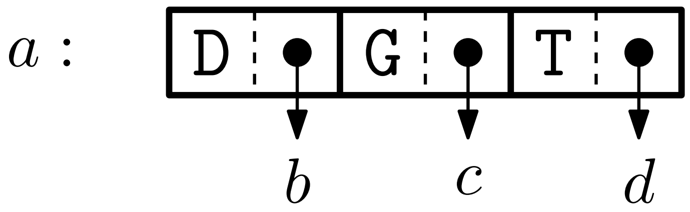
Time to find a symbol's edge/predecessor:
$O(\log \#\text{children}) = O(\log |\Sigma|)$

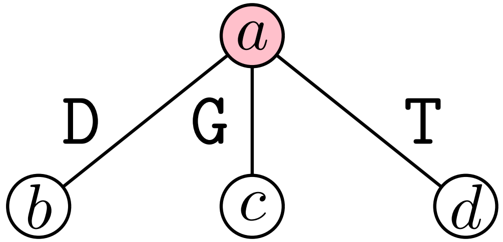
Representing Tries

$n = \#\text{nodes} = O\left(\sum_i |T_i|\right)$

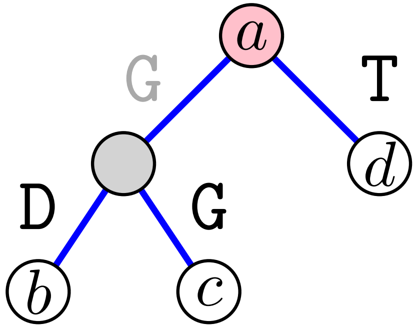
$\Sigma = \{\mathtt{A}, \mathtt{D}, \mathtt{E}, \mathtt{G}, \mathtt{R}, \mathtt{S}, \mathtt{T}\}$

Array (sparse)





Balanced Binary Search Tree



Space: $O(\#\text{children})$

Time to find a symbol's edge/predecessor:
$O(\log \#\text{children}) = O(\log |\Sigma|)$

Overall space: $O(n)$

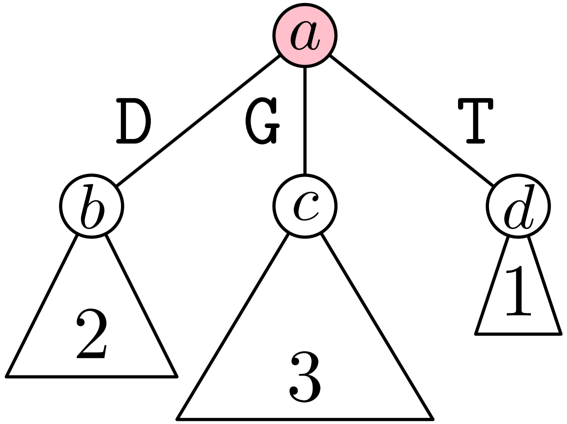
Overall time: $O(|P| \log |\Sigma|)$

Representing Tries

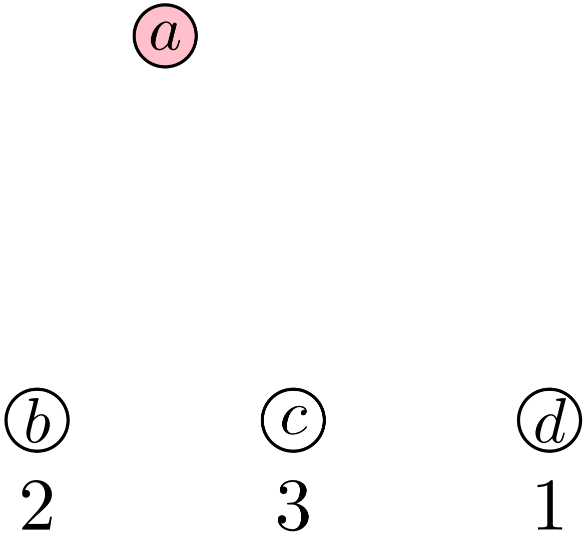
Weight-Balanced BSTs

$n = \#nodes = O\left(\sum_i |T_i|\right)$

Each vertex of the trie has a weight equal to the number of leaves in its subtree



Recursively construct a binary search tree by splitting the children in the trie so that the sum of their weights is as balanced as possible

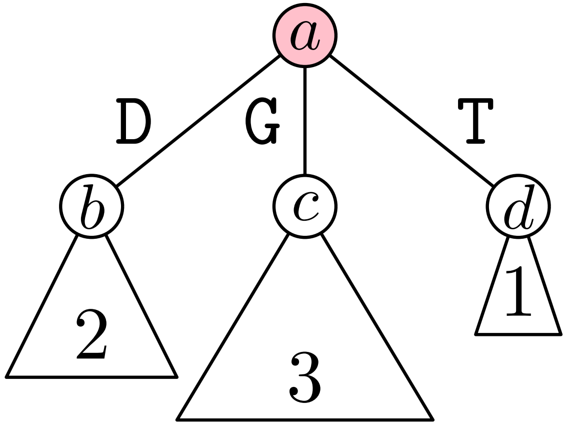


Representing Tries

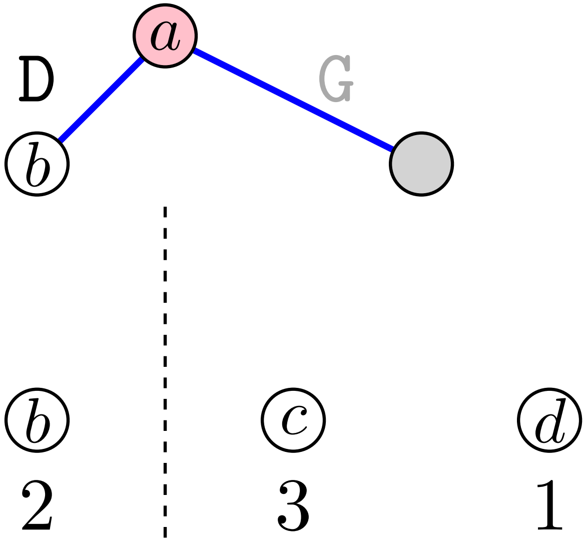
Weight-Balanced BSTs

$n = \#nodes = O\left(\sum_i |T_i|\right)$

Each vertex of the trie has a weight equal to the number of leaves in its subtree



Recursively construct a binary search tree by splitting the children in the trie so that the sum of their weights is as balanced as possible

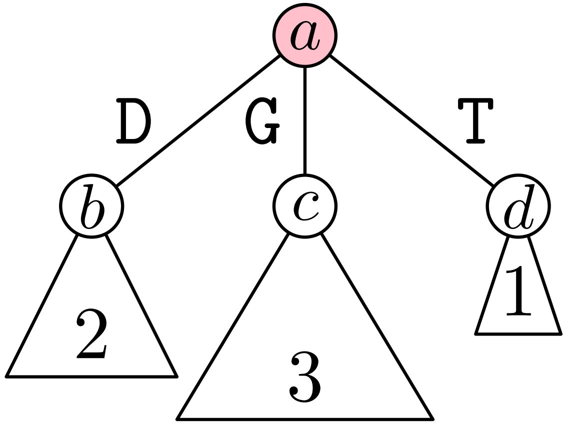


Representing Tries

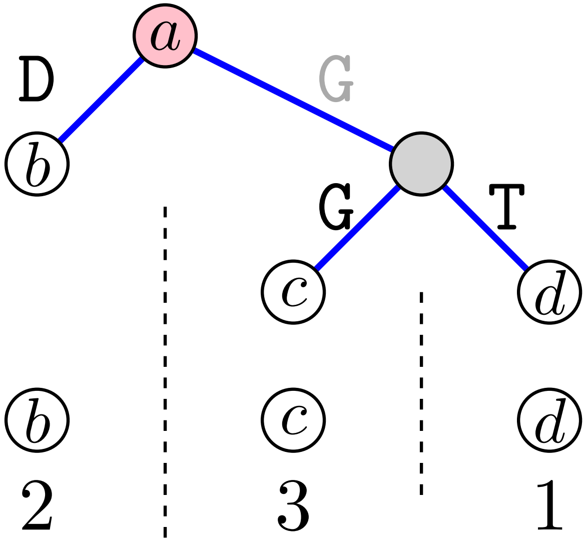
Weight-Balanced BSTs

$n = \#nodes = O\left(\sum_i |T_i|\right)$

Each vertex of the trie has a weight equal to the number of leaves in its subtree



Recursively construct a binary search tree by splitting the children in the trie so that the sum of their weights is as balanced as possible

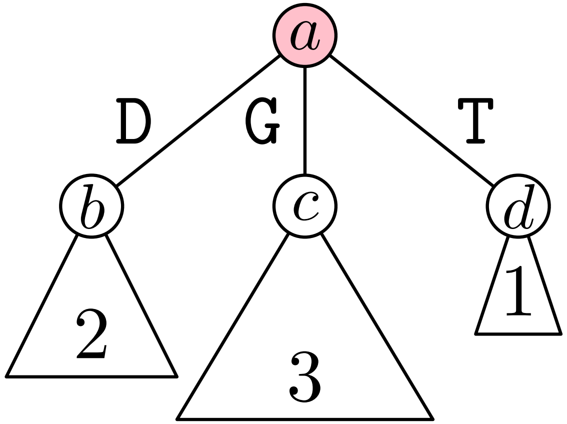


Representing Tries

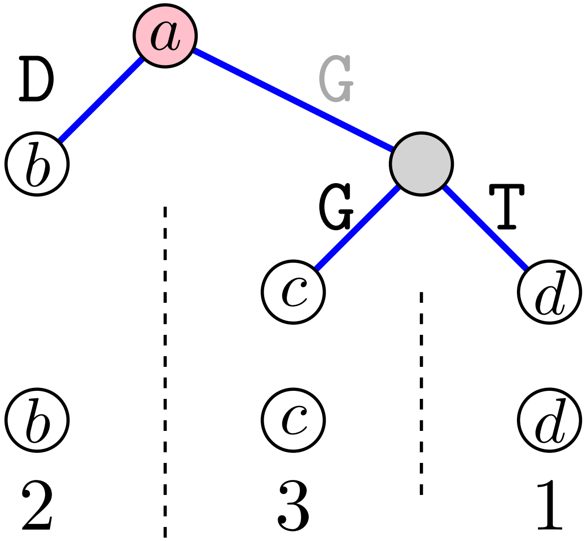
Weight-Balanced BSTs

$n = \#nodes = O\left(\sum_i |T_i|\right)$

Each vertex of the trie has a weight equal to the number of leaves in its subtree



Recursively construct a binary search tree by splitting the children in the trie so that the sum of their weights is as balanced as possible



Space: $O(\#\text{children})$

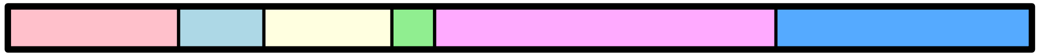
Overall space: $O(n)$

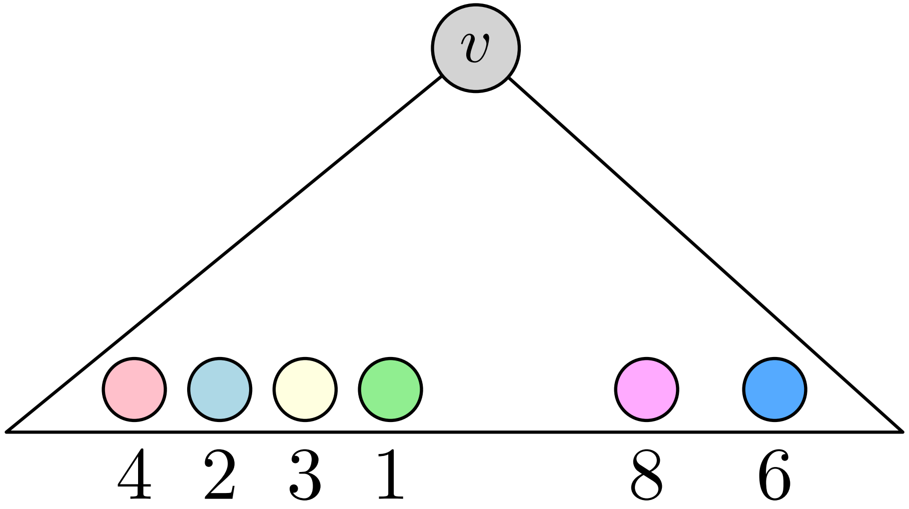
Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

Imagine the leaves in the subtree of v as consecutive segments with length equal to their weight





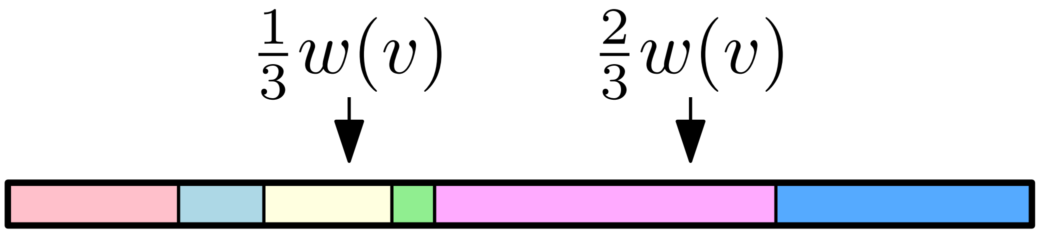
Representing Tries

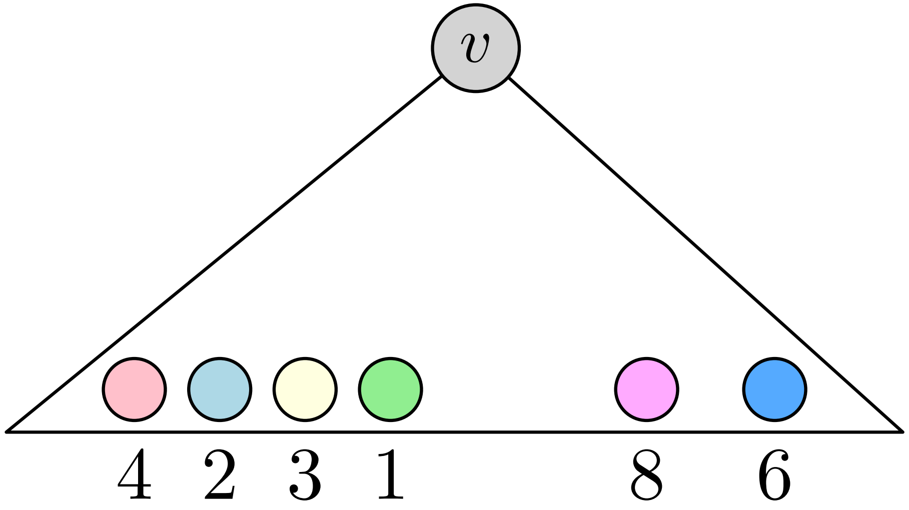
Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

Imagine the leaves in the subtree of v as consecutive segments with length equal to their weight

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains more than one segment:





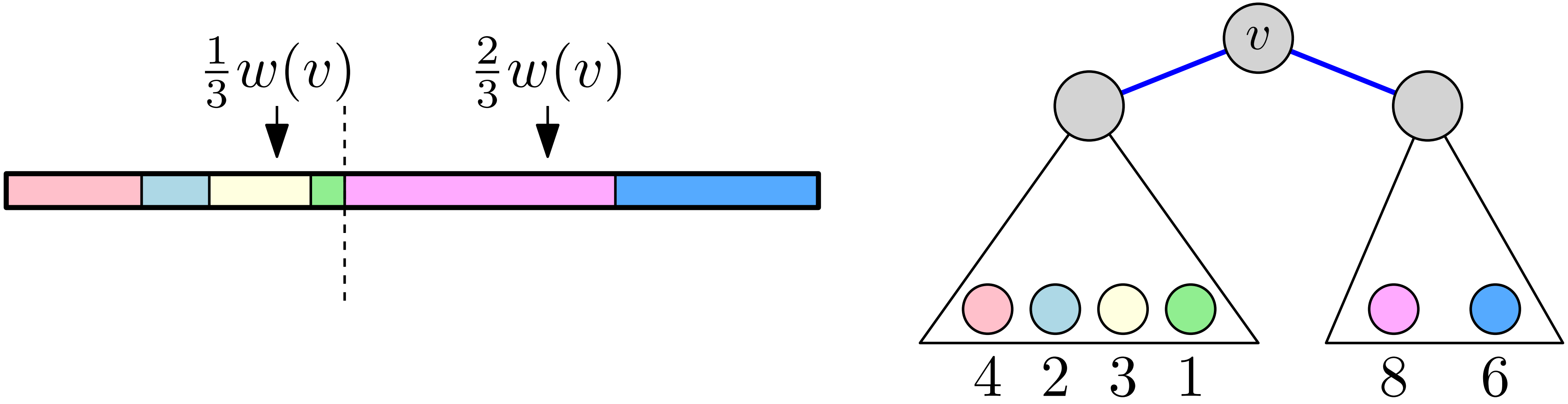
Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

Imagine the leaves in the subtree of v as consecutive segments with length equal to their weight

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains more than one segment:



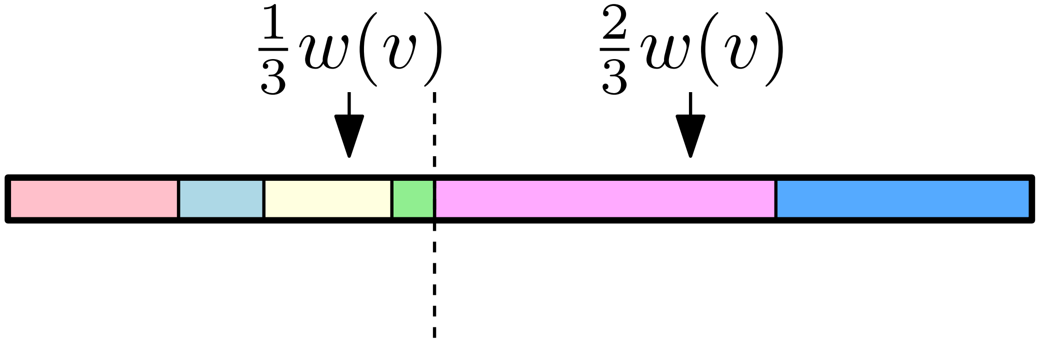
Representing Tries

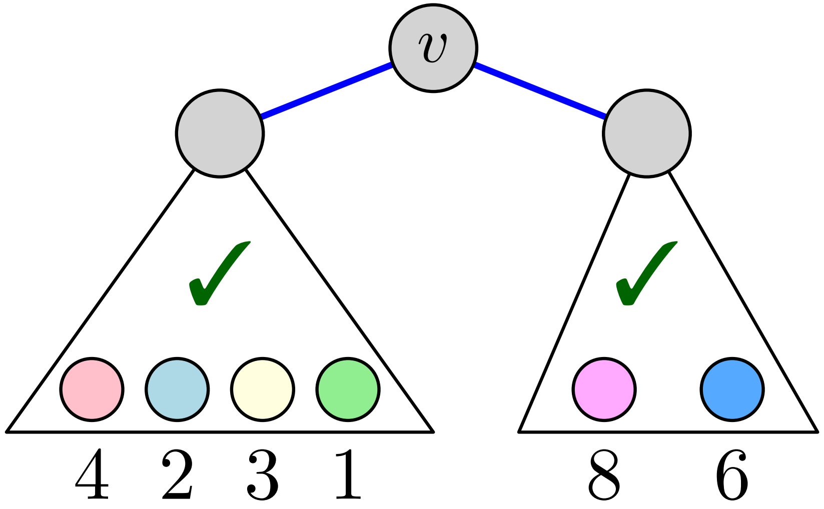
Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

Imagine the leaves in the subtree of v as consecutive segments with length equal to their weight

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains more than one segment:





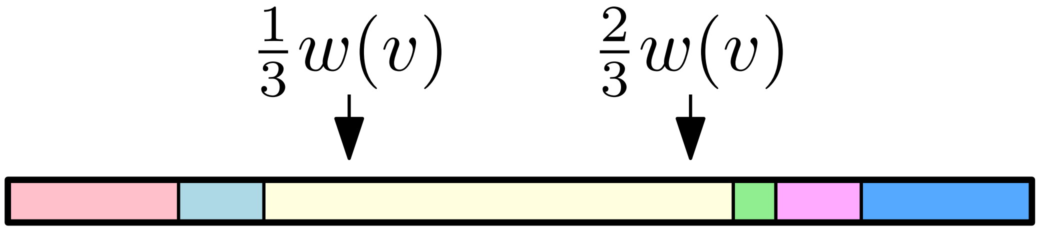
- the weight of each children of v is at most $\frac{2}{3}w(v)$

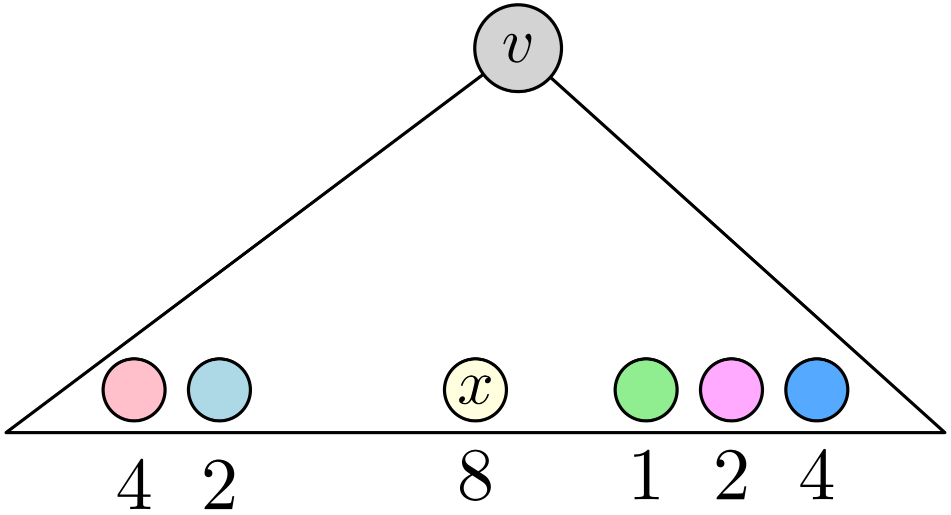
Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains a single segment, let x be the corresponding leaf



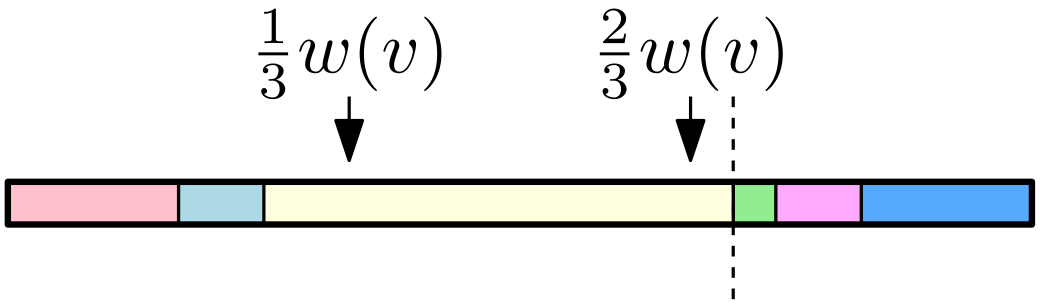


Representing Tries

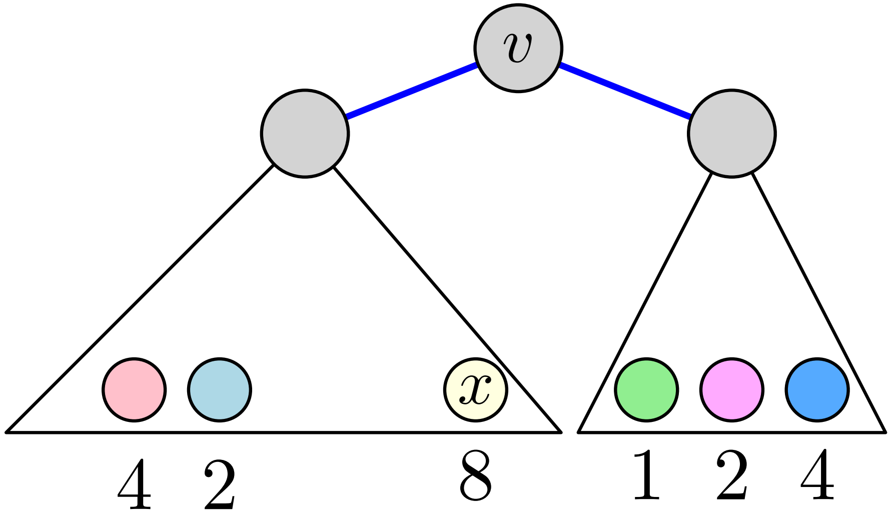
Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains a single segment, let x be the corresponding leaf



- v splits the segments immediately before/after x.

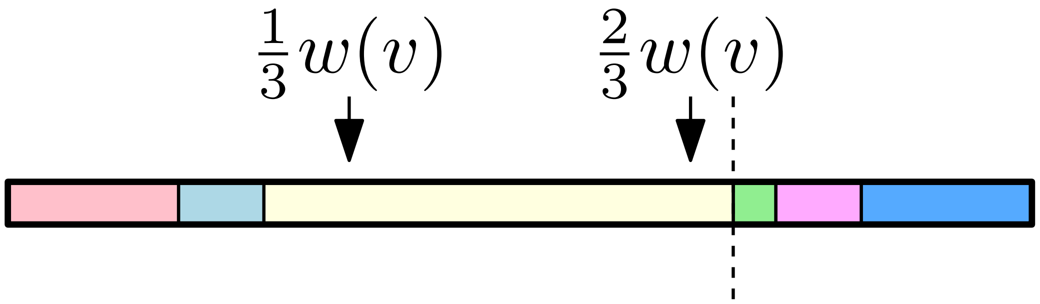


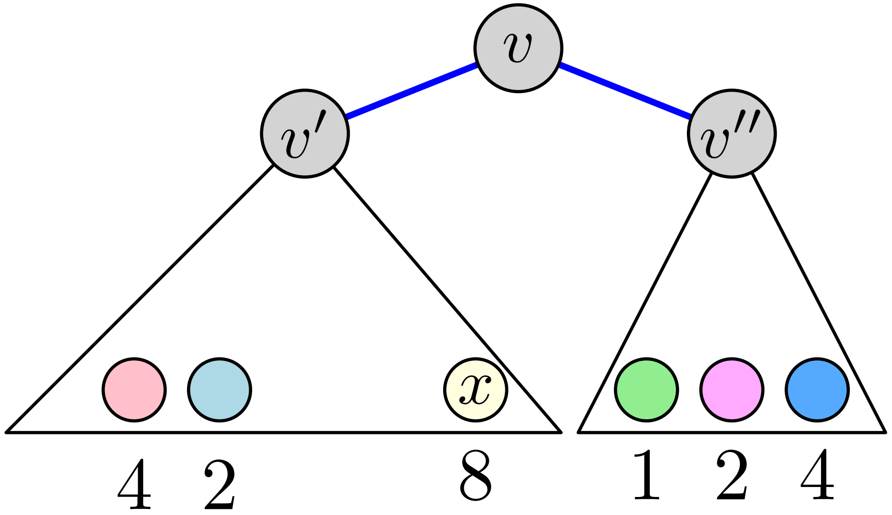
Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains a single segment, let x be the corresponding leaf





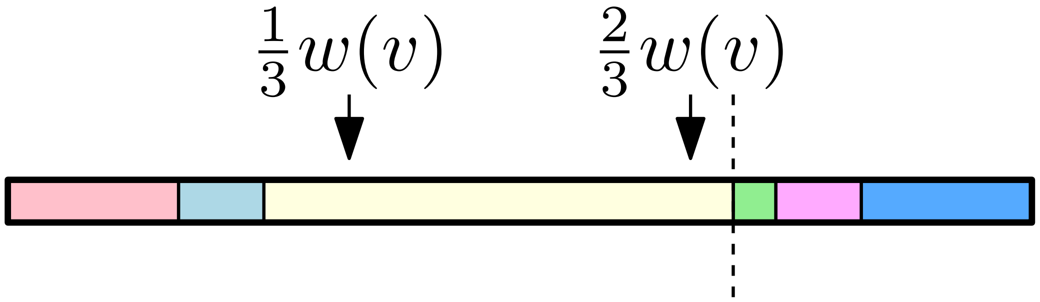
- v splits the segments immediately before/after x.
- Let v' be the child of v that contains x and let v'' be the other child

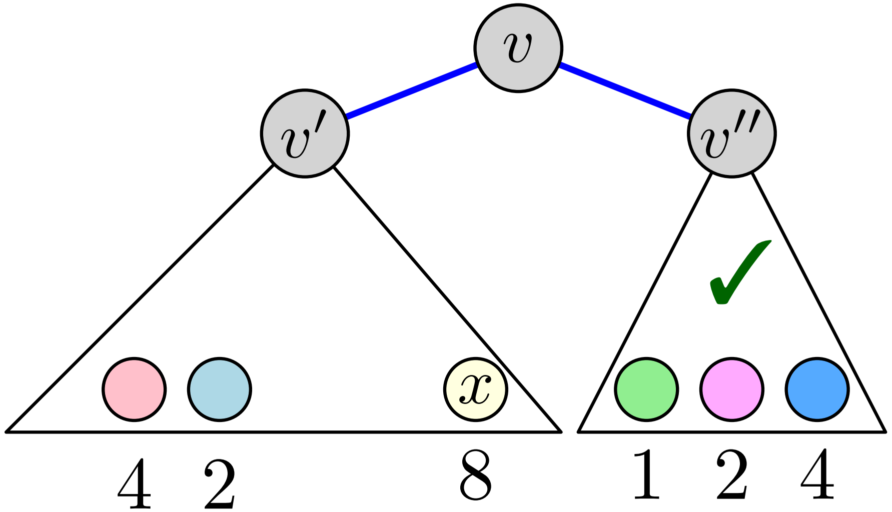
Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains a single segment, let x be the corresponding leaf





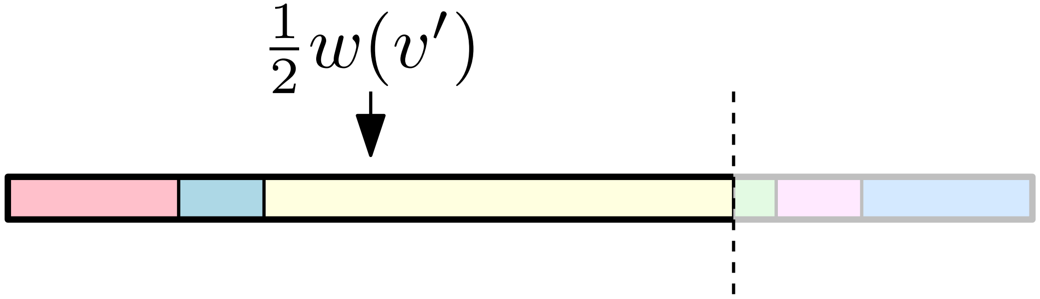
- v splits the segments immediately before/after x.
- Let v' be the child of v that contains x and let v'' be the other child
- $w(v'') \leq \frac{1}{3}w(v)$.

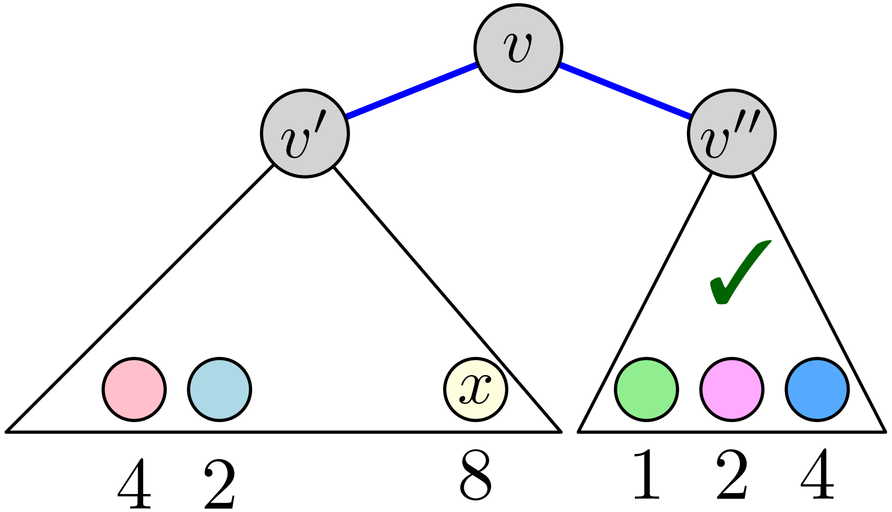
Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains a single segment, let x be the corresponding leaf





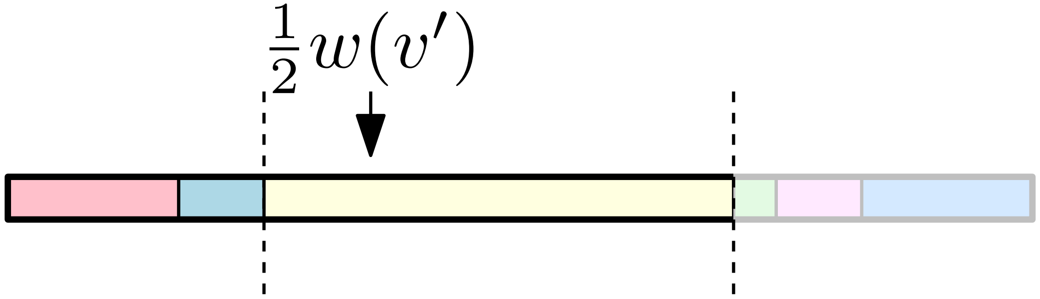
- v splits the segments immediately before/after x.
- Let v' be the child of v that contains x and let v'' be the other child
- $w(v'') \leq \frac{1}{3}w(v)$.
- x is the first or last leaf in the subtree of v' and $w(x) \geq \frac{1}{2}w(v')$

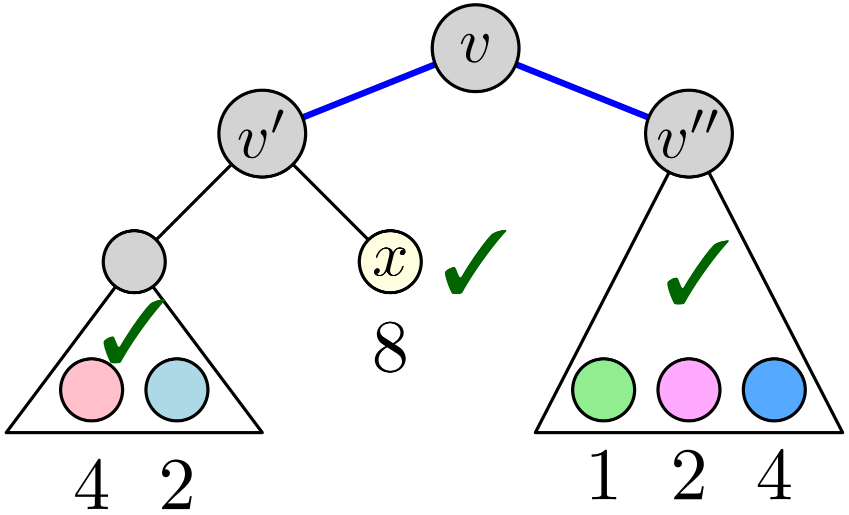
Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

If the interval $[\frac{1}{3}w(v), \frac{2}{3}w(v)]$ contains a single segment, let x be the corresponding leaf





- v splits the segments immediately before/after x.
- Let v' be the child of v that contains x and let v'' be the other child
- $w(v'') \leq \frac{1}{3}w(v)$.
- x is the first or last leaf in the subtree of v' and $w(x) \geq \frac{1}{2}w(v')$
- One child of v' is x and the other child weighs $\leq \frac{1}{2}w(v') \leq \frac{1}{2}w(v)$

Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

Traversing two edges of a weight-balanced BST either:

- Brings us to the next node in the trie, i.e., we advance one character into P; or

  Can only happen $O(|P|)$ times

Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

Traversing two edges of a weight-balanced BST either:

- Brings us to the next node in the trie, i.e., we advance one character into P; or

  Can only happen $O(|P|)$ times

- Reduces the weight (i.e, the number of leaves in the trie reachable from the current node) by $2/3$

  Can only happen $O(\log_{3/2} \#\text{leaves}) = O(\log k)$ times

Representing Tries

Weight-Balanced BSTs

Claim: All the grand-children u of v satisfy $w(u) \leq \frac{2}{3}(v)$ or are leaves.

Traversing two edges of a weight-balanced BST either:

- Brings us to the next node in the trie, i.e., we advance one character into P; or

  Can only happen $O(|P|)$ times

- Reduces the weight (i.e, the number of leaves in the trie reachable from the current node) by $2/3$

  Can only happen $O(\log_{3/2} \#\text{leaves}) = O(\log k)$ times

Overall space: $O(n)$

Overall time: $O(|P| + \log k)$

Representing Tries: Recap

	Space	**Query Time**
Array (dense)	$O(\lvert\Sigma\rvert \cdot n)$	$O(\lvert P\rvert)$
Array (sparse) / BST	$O(n)$	$O(\lvert P\rvert \log \lvert\Sigma\rvert)$
Weight-balanced BST	$O(n)$	$O(\lvert P\rvert + \log k)$

Representing Tries: Recap

	Space	**Query Time**
Array (dense)	$O(\lvert\Sigma\rvert \cdot n)$	$O(\lvert P\rvert)$
Array (sparse) / BST	$O(n)$	$O(\lvert P\rvert \log \lvert\Sigma\rvert)$
Weight-balanced BST	$O(\underline{n})$	$O(\underline{\lvert P\rvert} + \log k)$

Optimal

Representing Tries: Recap

	Space	**Query Time**
Array (dense)	$O(\lvert\Sigma\rvert \cdot n)$	$O(\lvert P\rvert)$
Array (sparse) / BST	$O(n)$	$O(\lvert P\rvert \log \lvert\Sigma\rvert)$
Weight-balanced BST	$O(\underline{n})$	$O(\underline{\lvert P\rvert} + \underline{\log k})$

Can we get rid of this term?

Optimal

Representing Tries: Recap

	Space	**Query Time**
Array (dense)	$O(\|\Sigma\| \cdot n)$	$O(\|P\|)$
Array (sparse) / BST	$O(n)$	$O(\|P\| \log \|\Sigma\|)$
Weight-balanced BST	$O(\underline{n})$	$O(\underline{\|P\|} + \underline{\log k})$

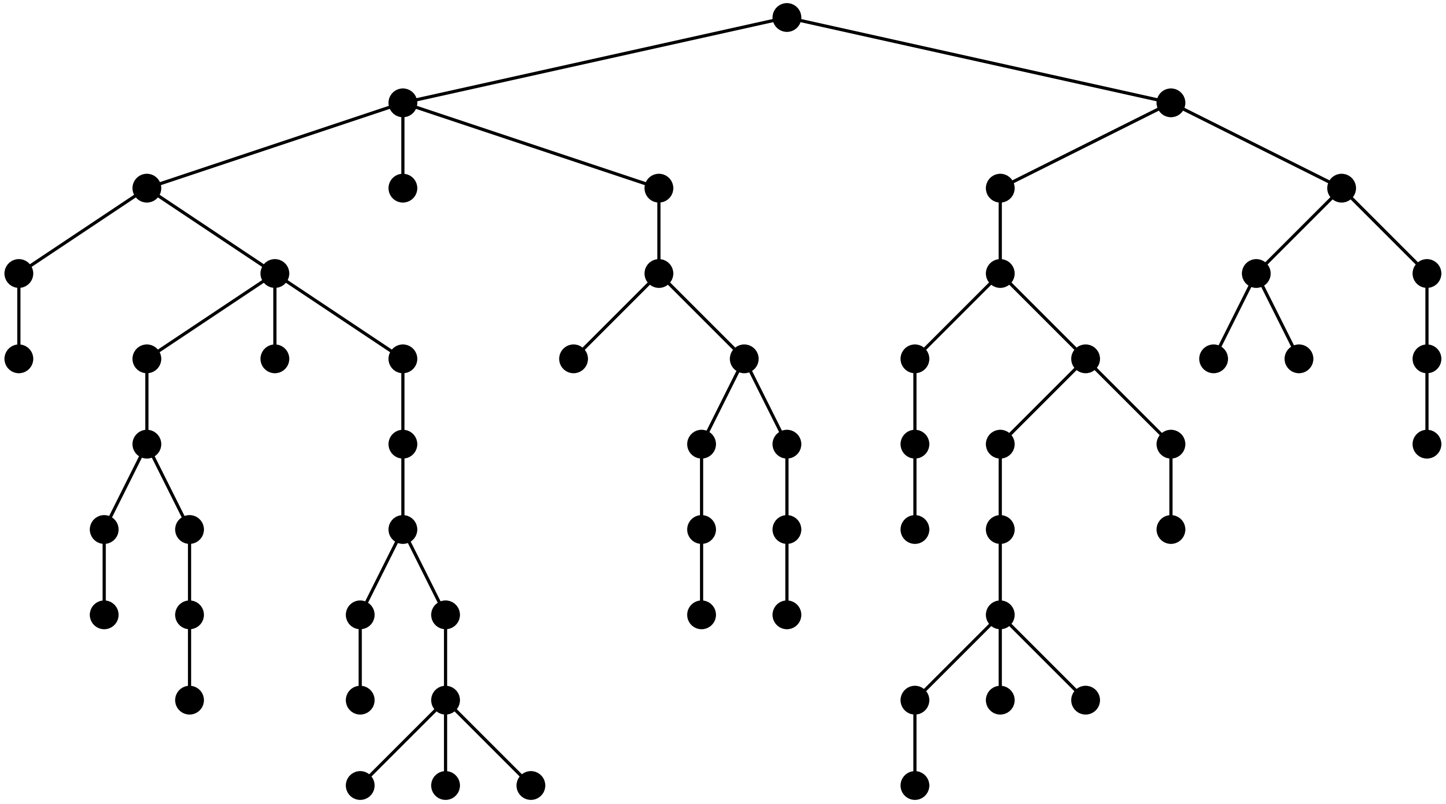
Can we get rid of this term?

Almost...

Optimal

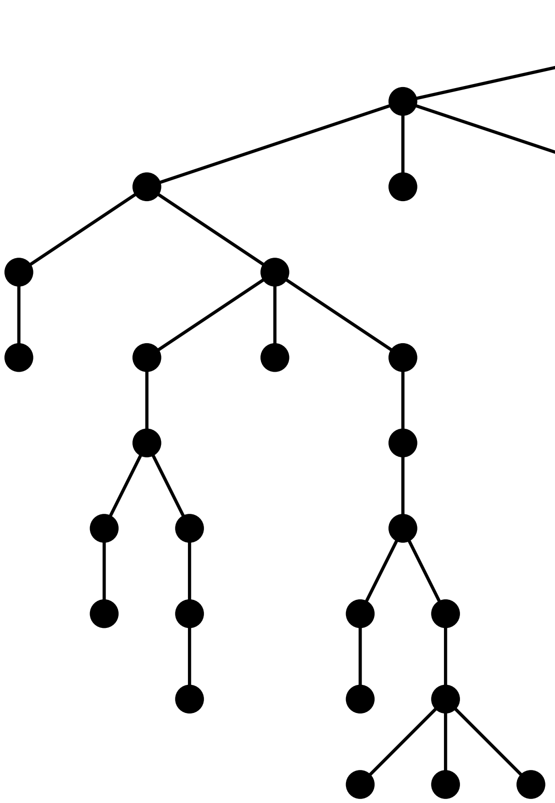
Indirection

We can use a similar technique to the one we encountered while designing level ancestor oracles



Indirection

We can use a similar technique to the one we encountered while designing level ancestor oracles

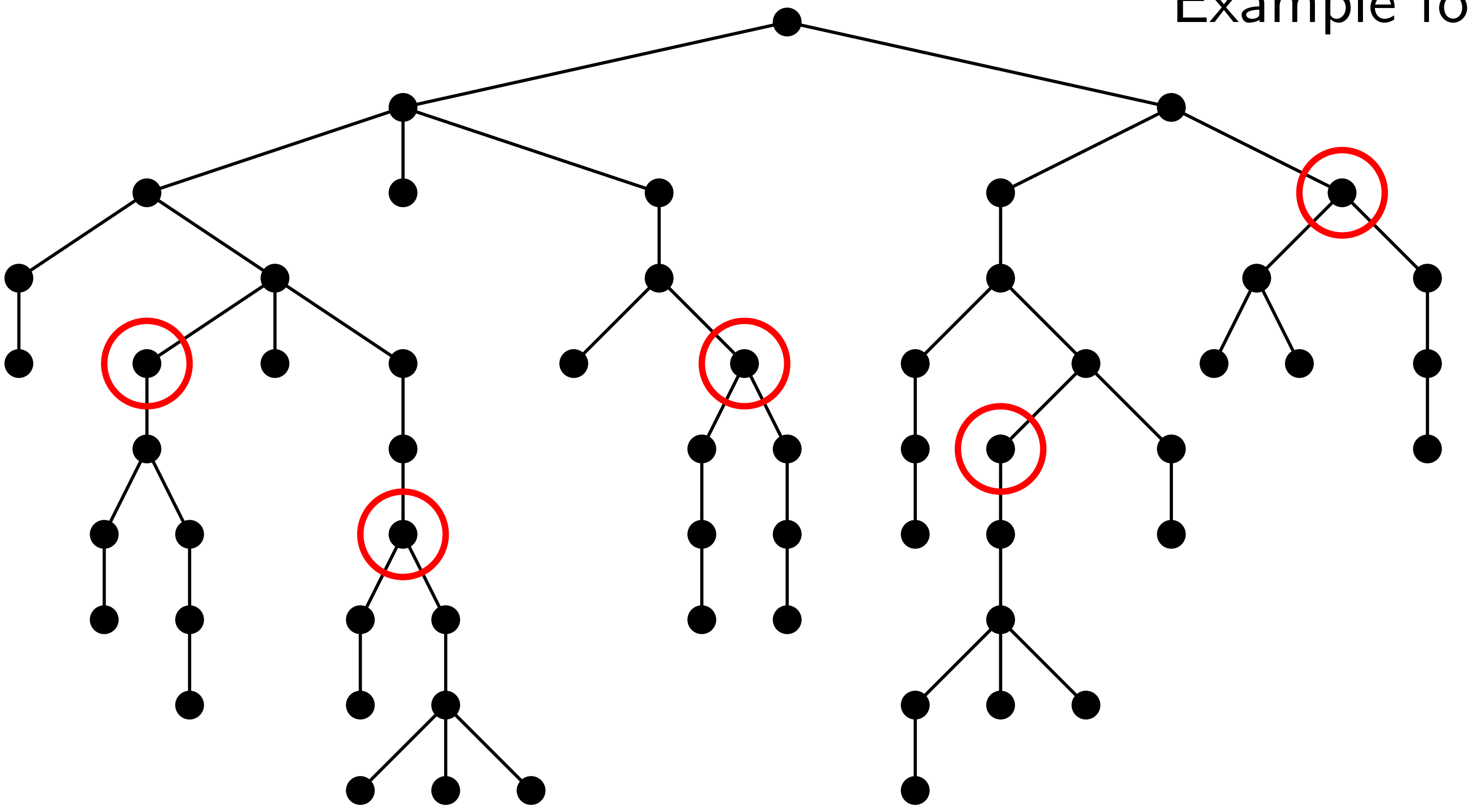


Find the set M of all maximally deep vertices with at least $|\Sigma|$ descendants

Indirection

We can use a similar technique to the one we encountered while designing level ancestor oracles

Example for $|\Sigma| = 7$

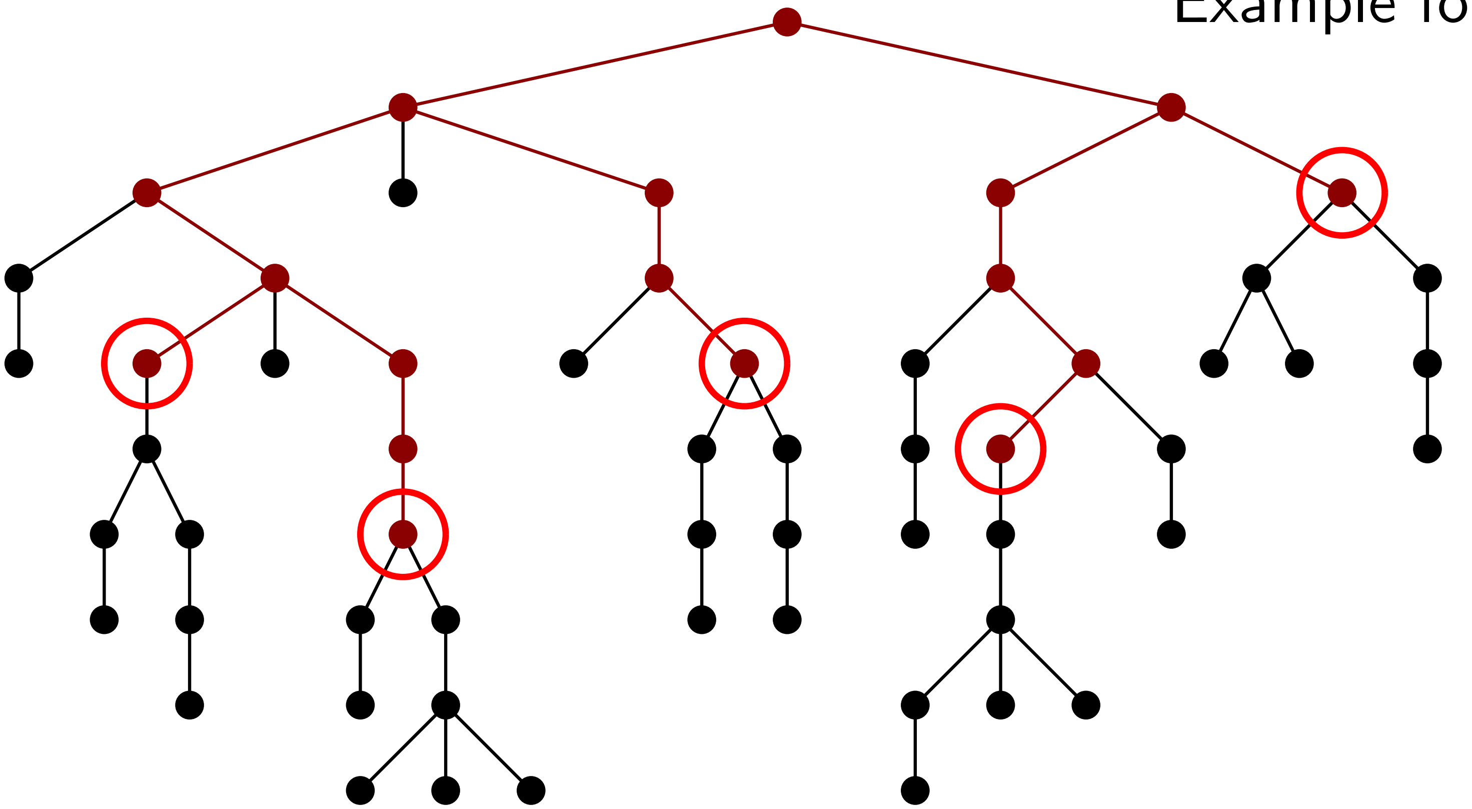


Find the set M of all maximally deep vertices with at least $|\Sigma|$ descendants

Indirection

We can use a similar technique to the one we encountered while designing level ancestor oracles

Example for $|\Sigma| = 7$



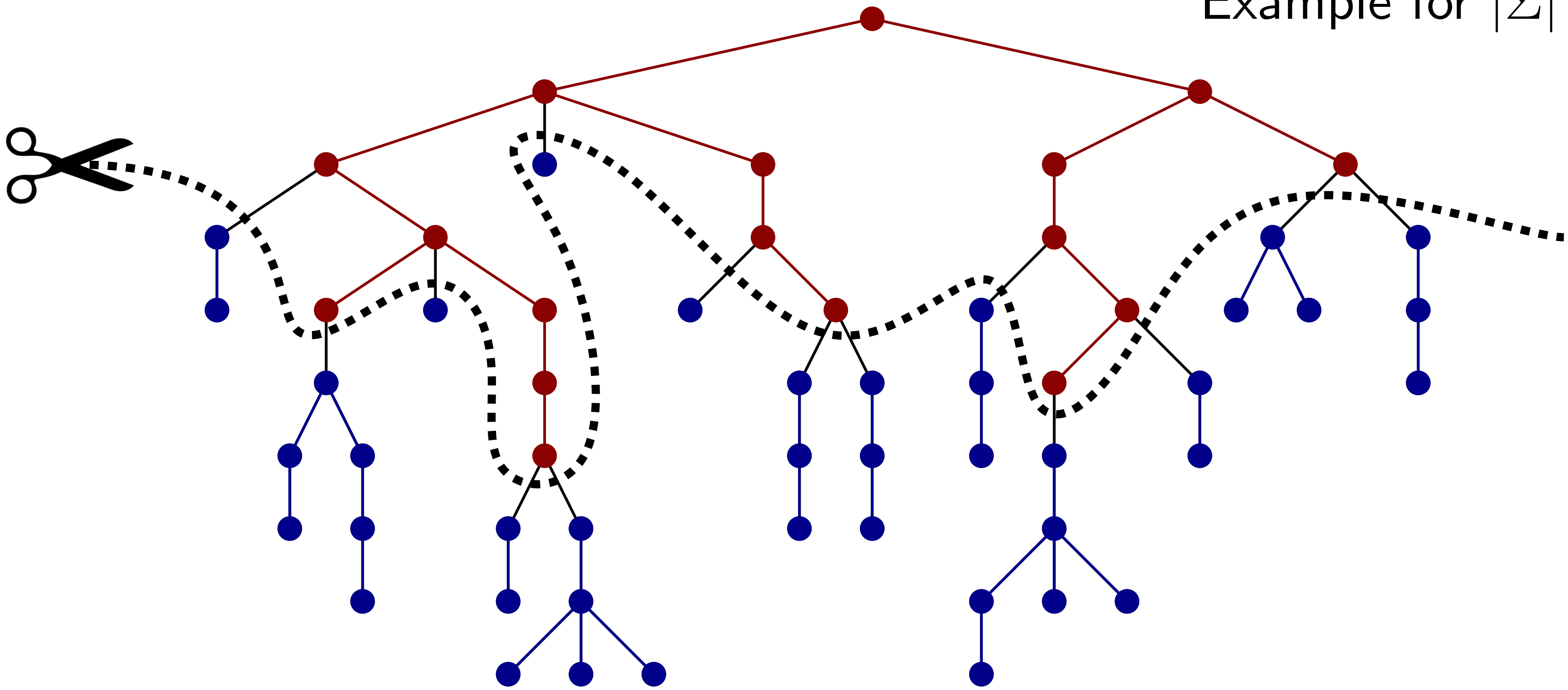
Find the set M of all maximally deep vertices with at least $|\Sigma|$ descendants

Split the trie into a tree T' containing all the ancestors of the vertices in M and several bottom-trees in $T \setminus T'$.

Indirection

We can use a similar technique to the one we encountered while designing level ancestor oracles

Example for $|\Sigma| = 7$



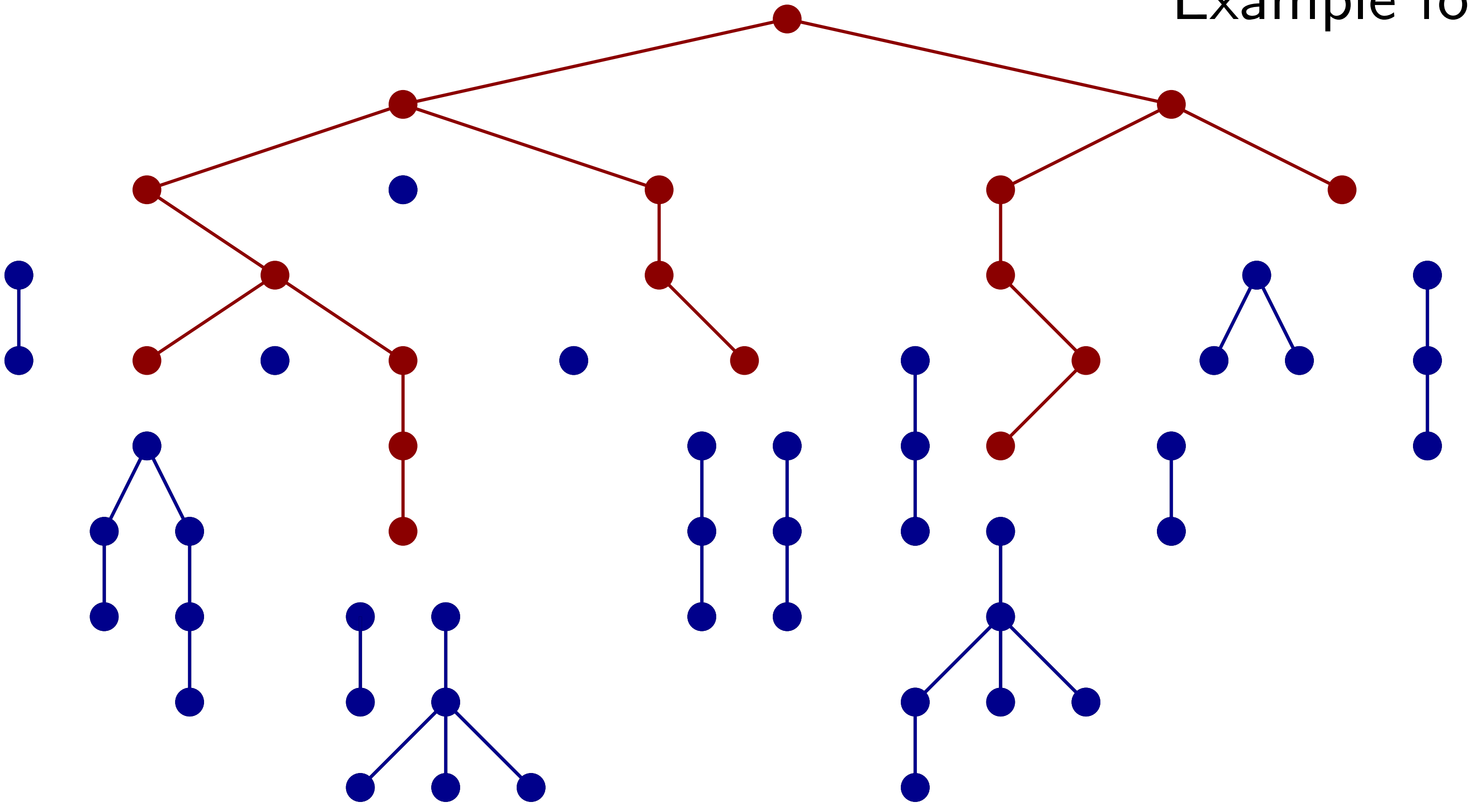
Find the set M of all maximally deep vertices with at least $|\Sigma|$ descendants

Split the trie into a tree T' containing all the ancestors of the vertices in M and several bottom-trees in $T \setminus T'$.

Indirection

We can use a similar technique to the one we encountered while designing level ancestor oracles

Example for $|\Sigma| = 7$



Find the set M of all maximally deep vertices with at least $|\Sigma|$ descendants

Split the trie into a tree T' containing all the ancestors of the vertices in M and several bottom-trees in $T \setminus T'$.

Indirection

Storing the top tree:

The number of leaves of T' is at most $\frac{n}{|\Sigma|}$

Fact: A tree with ℓ leaves has at most $\ell - 1$ branching nodes (i.e., nodes with at least 2 children)

Indirection

Storing the top tree:

The number of leaves of T' is at most $\frac{n}{|\Sigma|}$

Fact: A tree with ℓ leaves has at most $\ell - 1$ branching nodes (i.e., nodes with at least 2 children)

- Store leaves using dense arrays

Space

$O(|\Sigma| \cdot \frac{n}{|\Sigma|}) = O(n)$

Indirection

Storing the top tree:

The number of leaves of T' is at most $\frac{n}{|\Sigma|}$

Fact: A tree with ℓ leaves has at most $\ell - 1$ branching nodes (i.e., nodes with at least 2 children)

Space

- Store leaves using dense arrays $O(|\Sigma| \cdot \frac{n}{|\Sigma|}) = O(n)$
- Store branching nodes using dense arrays $O(|\Sigma| \cdot \frac{n}{|\Sigma|}) = O(n)$

Indirection

Storing the top tree:

The number of leaves of T' is at most $\frac{n}{|\Sigma|}$

Fact: A tree with ℓ leaves has at most $\ell - 1$ branching nodes (i.e., nodes with at least 2 children)

	Space
• Store leaves using dense arrays	$O(\lvert\Sigma\rvert \cdot \frac{n}{\lvert\Sigma\rvert}) = O(n)$
• Store branching nodes using dense arrays	$O(\lvert\Sigma\rvert \cdot \frac{n}{\lvert\Sigma\rvert}) = O(n)$
• Store the unique child of each non-branching nodes explicitly	$O(n)$

Indirection

Storing the top tree:

The number of leaves of T' is at most $\frac{n}{|\Sigma|}$

Fact: A tree with ℓ leaves has at most $\ell - 1$ branching nodes (i.e., nodes with at least 2 children)

	Space
• Store leaves using dense arrays	$O(\lvert\Sigma\rvert \cdot \frac{n}{\lvert\Sigma\rvert}) = O(n)$
• Store branching nodes using dense arrays	$O(\lvert\Sigma\rvert \cdot \frac{n}{\lvert\Sigma\rvert}) = O(n)$
• Store the unique child of each non-branching nodes explicitly	$O(n)$

Time to find the next node $O(1)$

Indirection

Storing the top tree:

The number of leaves of T' is at most $\frac{n}{|\Sigma|}$

Fact: A tree with ℓ leaves has at most $\ell - 1$ branching nodes (i.e., nodes with at least 2 children)

Space

- Store leaves using dense arrays $O(|\Sigma| \cdot \frac{n}{|\Sigma|}) = O(n)$
- Store branching nodes using dense arrays $O(|\Sigma| \cdot \frac{n}{|\Sigma|}) = O(n)$
- Store the unique child of each non-branching nodes explicitly $O(n)$

Time to find the next node $O(1)$

Storing the bottom trees:

- Store each bottom tree using a weight-balanced BST

Total space of all bottom trees: $O(n)$

Indirection

Storing the top tree:

The number of leaves of T' is at most $\frac{n}{|\Sigma|}$

Fact: A tree with ℓ leaves has at most $\ell - 1$ branching nodes (i.e., nodes with at least 2 children)

Space

- Store leaves using dense arrays $O(|\Sigma| \cdot \frac{n}{|\Sigma|}) = O(n)$
- Store branching nodes using dense arrays $O(|\Sigma| \cdot \frac{n}{|\Sigma|}) = O(n)$
- Store the unique child of each non-branching nodes explicitly $O(n)$

Time to find the next node $O(1)$

Storing the bottom trees:

- Store each bottom tree using a weight-balanced BST

Total space of all bottom trees: $O(n)$

- Each bottom tree has at most $|\Sigma|$ leaves

Time to navigate a bottom tree: $O(|P| + \log|\Sigma|)$

Representing Tries: Recap

	Space	**Query Time**
Array (dense)	$O(\lvert\Sigma\rvert \cdot n)$	$O(\lvert P\rvert)$
Array (sparse) / BST	$O(n)$	$O(\lvert P\rvert \log \lvert\Sigma\rvert)$
Weight-balanced BST	$O(n)$	$O(\lvert P\rvert + \log k)$
Indirection	$O(n)$	$O(\lvert P\rvert + \log \lvert\Sigma\rvert)$

Can be made dynamic with a time complexity of $O(|T| + \log|\Sigma|)$ per insertion/deletion of T

Application: String Sorting

Sort a collection of k strings $T_1, T_2, \ldots, T_k$ over Σ

$L = \max_{i=1,\ldots,k} |T_i|$

Obs: A string comparison requires time $O(L)$.
Naive sorting algorithm take time $O(Lk \log k)$ or $O(Lk)$

Application: String Sorting

Sort a collection of k strings $T_1, T_2, \ldots, T_k$ over Σ

$L = \max_{i=1,\ldots,k} |T_i|$

Obs: A string comparison requires time $O(L)$.
Naive sorting algorithm take time $O(Lk \log k)$ or $O(Lk)$

- Create an empty trie
- For $i = 1, \ldots, k$:
    - Insert T_i into the trie
- An in-order visit of the trie returns the strings in lexicographic order

Application: String Sorting

Sort a collection of k strings $T_1, T_2, \ldots, T_k$ over Σ

$L = \max_{i=1,\ldots,k} |T_i|$

Obs: A string comparison requires time $O(L)$.
Naive sorting algorithm take time $O(Lk \log k)$ or $O(Lk)$

Time

- Create an empty trie
- For $i = 1, \ldots, k$:
  - Insert T_i into the trie

$$\left.\begin{array}{l} \\ \\ \end{array}\right\} O\left(\sum_{i=1}^{k} (|T_i| + \log|\Sigma|)\right)$$

- An in-order visit of the trie returns the strings in lexicographic order

Application: String Sorting

Sort a collection of k strings $T_1, T_2, \ldots, T_k$ over Σ

$L = \max_{i=1,\ldots,k} |T_i|$

Obs: A string comparison requires time $O(L)$.
Naive sorting algorithm take time $O(Lk \log k)$ or $O(Lk)$

Time

- Create an empty trie
- For $i = 1, \ldots, k$:
  - Insert T_i into the trie

$\left.\right\} O\left(n + k \log |\Sigma|\right)$

- An in-order visit of the trie returns the strings in lexicographic order

Application: String Sorting

Sort a collection of k strings $T_1, T_2, \ldots, T_k$ over Σ

$$L = \max_{i=1,\ldots,k} |T_i|$$

Obs: A string comparison requires time $O(L)$.
Naive sorting algorithm take time $O(Lk \log k)$ or $O(Lk)$

Time

- Create an empty trie
- For $i = 1, \ldots, k$:
  - Insert T_i into the trie

$$\Bigg\} \; O\left(n + k \log |\Sigma|\right)$$

- An in-order visit of the trie returns the strings in lexicographic order $\quad O(n)$

Application: String Sorting

Sort a collection of k strings $T_1, T_2, \ldots, T_k$ over Σ

$L = \max_{i=1,\ldots,k} |T_i|$

Obs: A string comparison requires time $O(L)$.
Naive sorting algorithm take time $O(Lk \log k)$ or $O(Lk)$

Time

- Create an empty trie
- For $i = 1, \ldots, k$:
  - Insert T_i into the trie

$\left.\right\} O\left(n + k \log |\Sigma|\right)$

- An in-order visit of the trie returns the strings in lexicographic order $O(n)$

Overall time: $O\left(n + k \log |\Sigma|\right)$

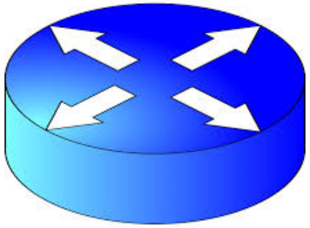
Application: Packet Routing

Among all the destinations that match, a packet gets routed to the one with the most specific rule

Packet

Src: 192.168.42.10

Dst: 101.167.200.15



Routing Table

Destination	Interface
169.0.0.0/11	eth1
169.48.0.0/12	ppp0
169.128.0.0/10	eth1
169.160.0.0/11	eth0
96.0.0.0/3	tun1
96.0.0.0/5	tun0
100.0.0.0/8	eth0
127.0.0.0/8	lo
default	wlan0

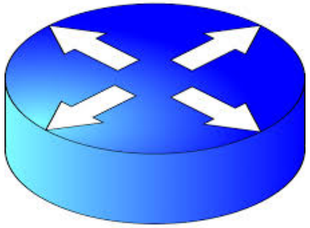
Application: Packet Routing

Among all the destinations that match, a packet gets routed to the one with the most specific rule

Packet

Src: 192.168.42.10

Dst: 0110010110100111. . .



Routing Table

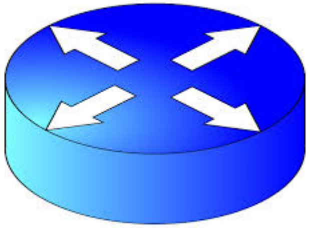
Destination	Interface
10101001000$	eth1
101010010011$	ppp0
1010100110$	eth1
10101001101$	eth0
011$	tun1
011000$	tun0
01100100$	eth0
01111111$	lo
$	wlan0

Application: Packet Routing

Among all the destinations that match, a packet gets routed to the one with the most specific rule

Packet

Src:	192.168.42.10
Dst:	011001011010011 1. . .



Routing Table

Destination	Interface
10101001000$	eth1
101010010011$	ppp0
1010100110$	eth1
10101001101$	eth0
011$	tun1
011000$	tun0
01100100$	eth0
01111111$	lo
$	wlan0

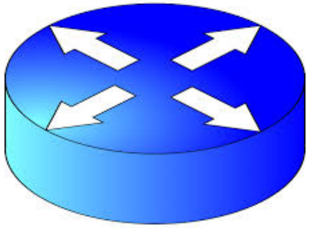
Application: Packet Routing

Among all the destinations that match, a packet gets routed to the one with the most specific rule

Packet

Src: 192.168.42.10

Dst: $\underbrace{0110010110100111\ldots}_{P}$



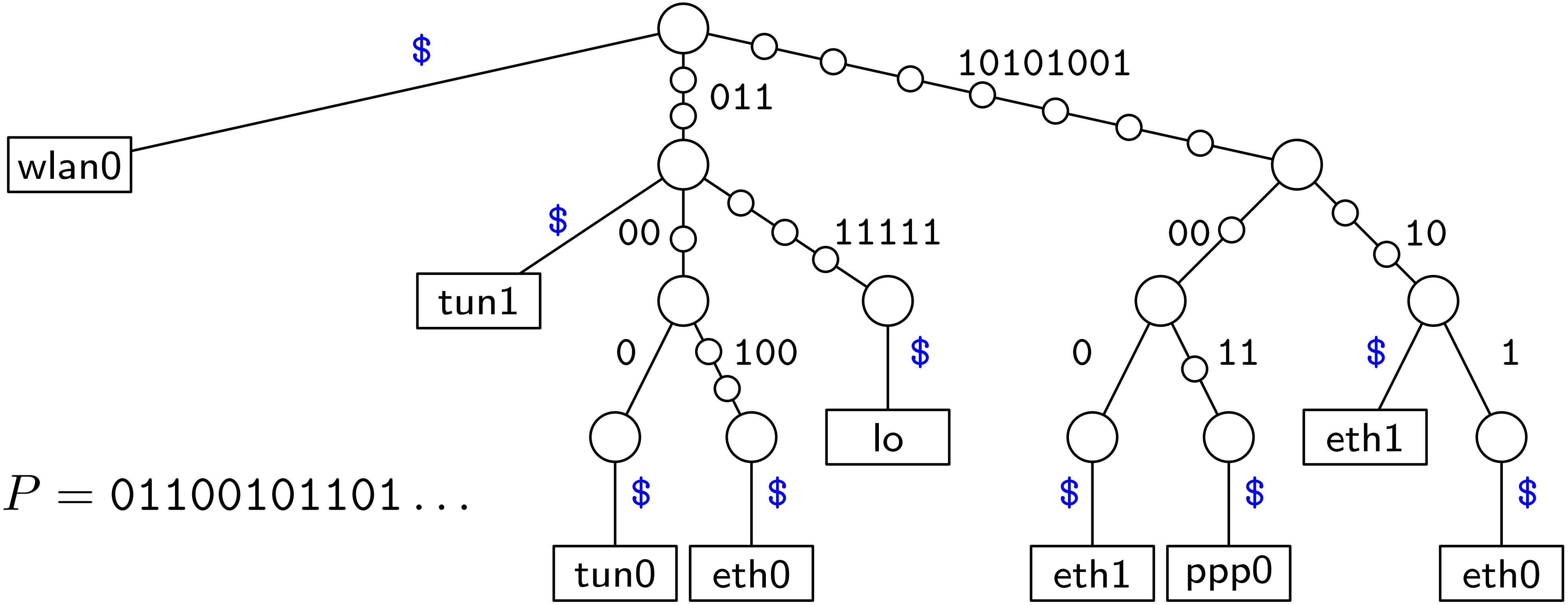
Routing Table

Destination	Interface
10101001000$	eth1
101010010011$	ppp0
1010100110$	eth1
10101001101$	eth0
011$	tun1
011000$	tun0
01100100$	eth0
01111111$	lo
$	wlan0

Given a pattern P we want the longest string in our collection that appears as a prefix of P

Application: Packet Routing

Build a trie T with all the addresses in the routing table.

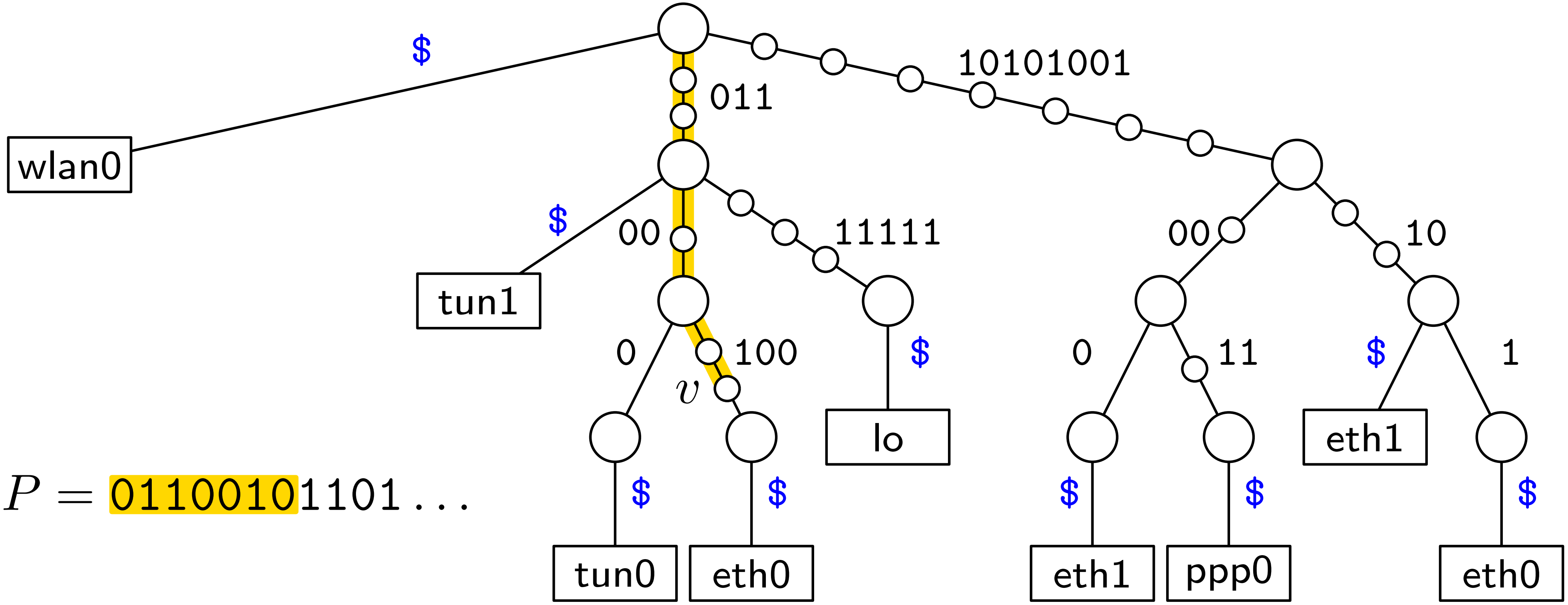


$P = 01100101101\ldots$

- Find the node v corresponding to the maximal prefix that matches P

Application: Packet Routing

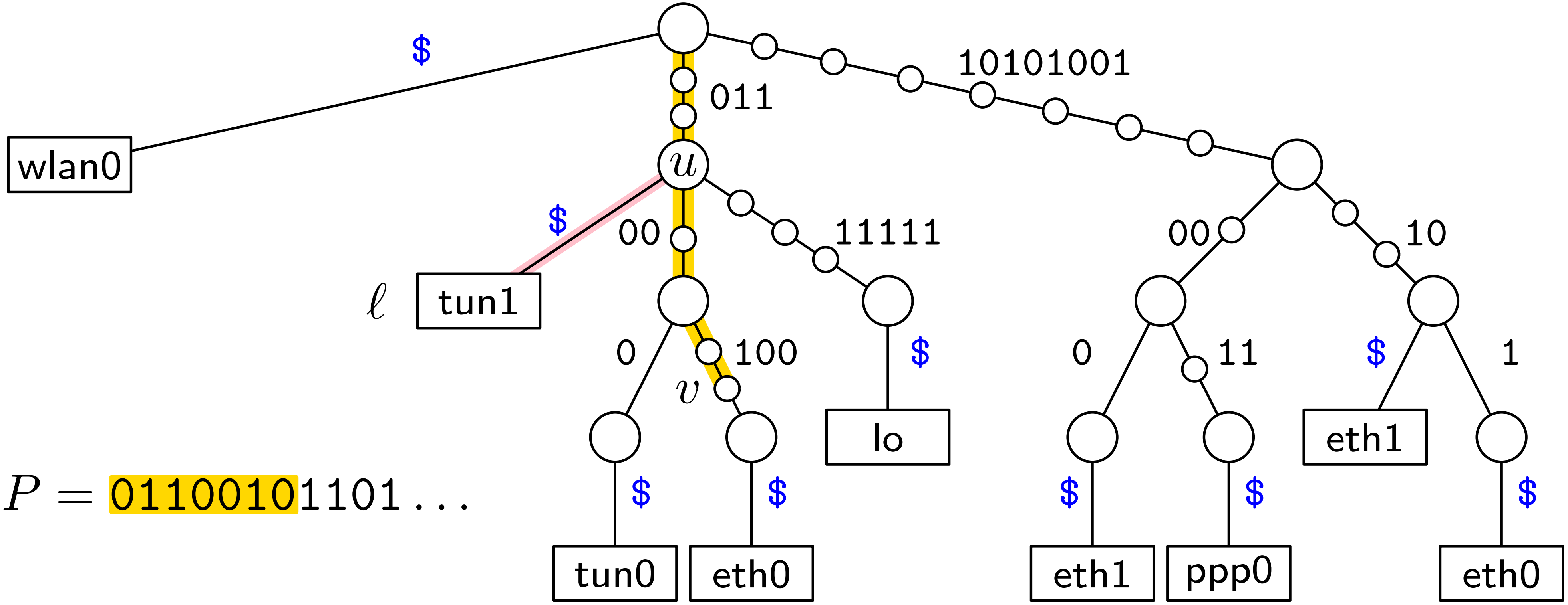
Build a trie T with all the addresses in the routing table.



- Find the node v corresponding to the maximal prefix that matches P

Application: Packet Routing

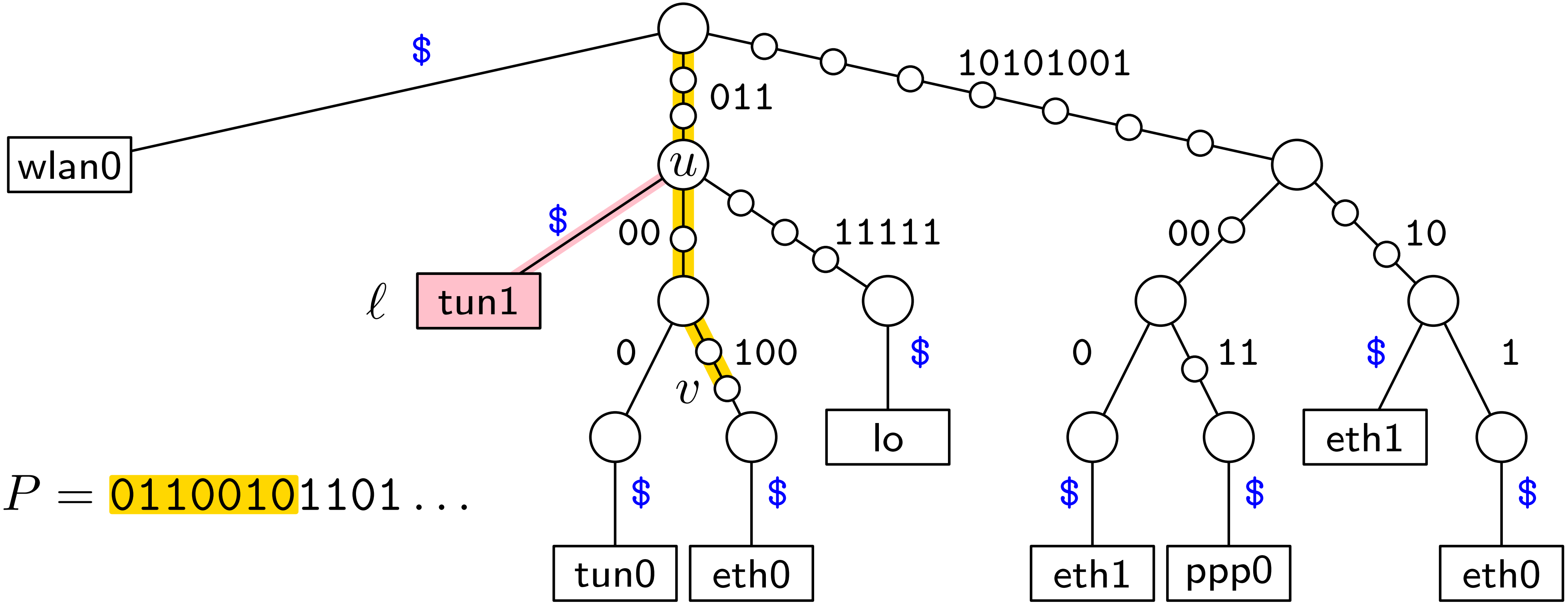
Build a trie T with all the addresses in the routing table.



- Find the node v corresponding to the maximal prefix that matches P
- Walk up the tree searching for the deepest ancestor u of v incident to a "$" edge towards a leaf ℓ

Application: Packet Routing

Build a trie T with all the addresses in the routing table.

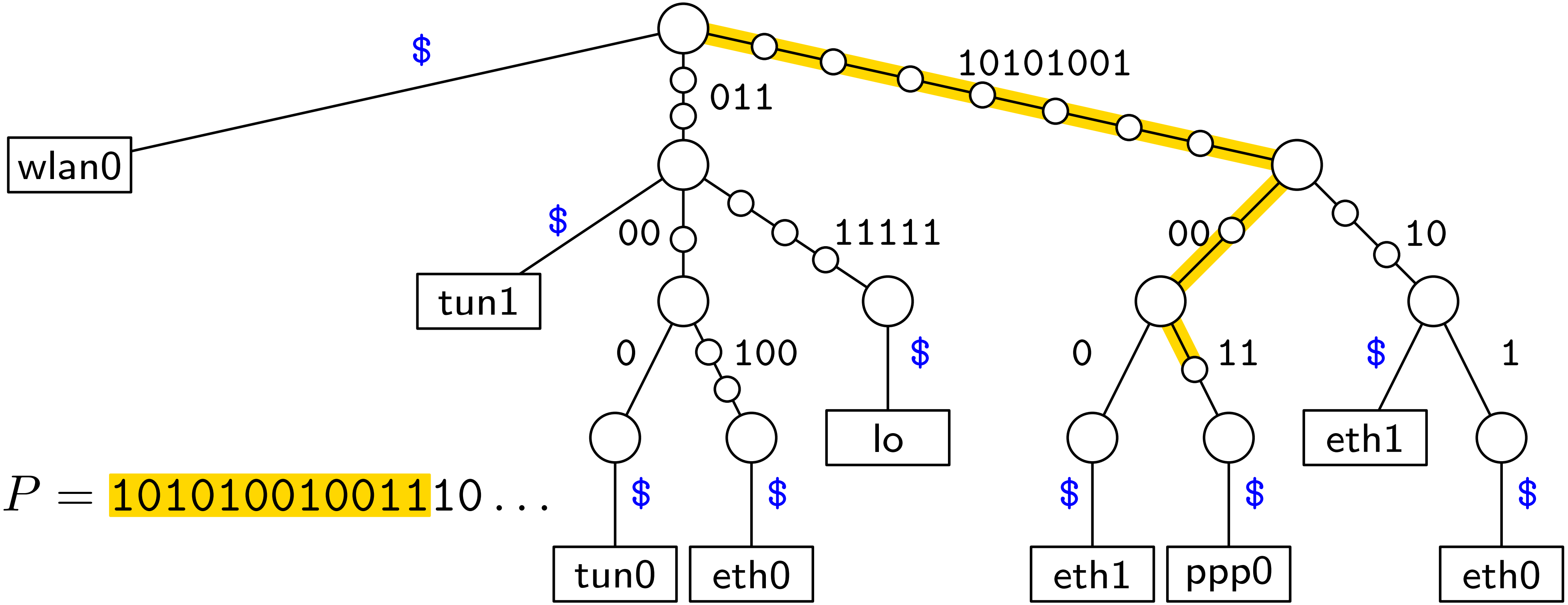


$P = 01100101101\ldots$

- Find the node v corresponding to the maximal prefix that matches P
- Walk up the tree searching for the deepest ancestor u of v incident to a "$" edge towards a leaf ℓ
- Route the packet towards the interface stored in ℓ

Application: Packet Routing

Build a trie T with all the addresses in the routing table.

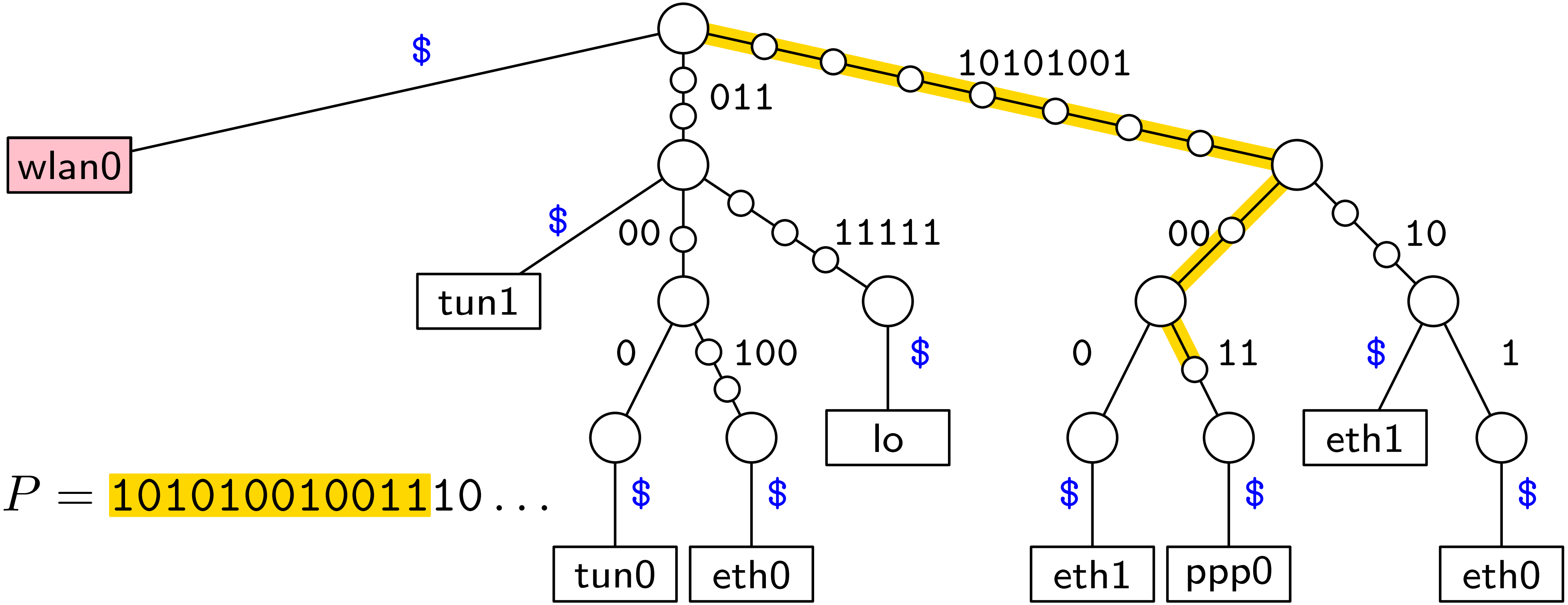


$P = 10101001001110\dots$

- Find the node v corresponding to the maximal prefix that matches P
- Walk up the tree searching for the deepest ancestor u of v incident to a "$" edge towards a leaf ℓ
- Route the packet towards the interface stored in ℓ

Application: Packet Routing

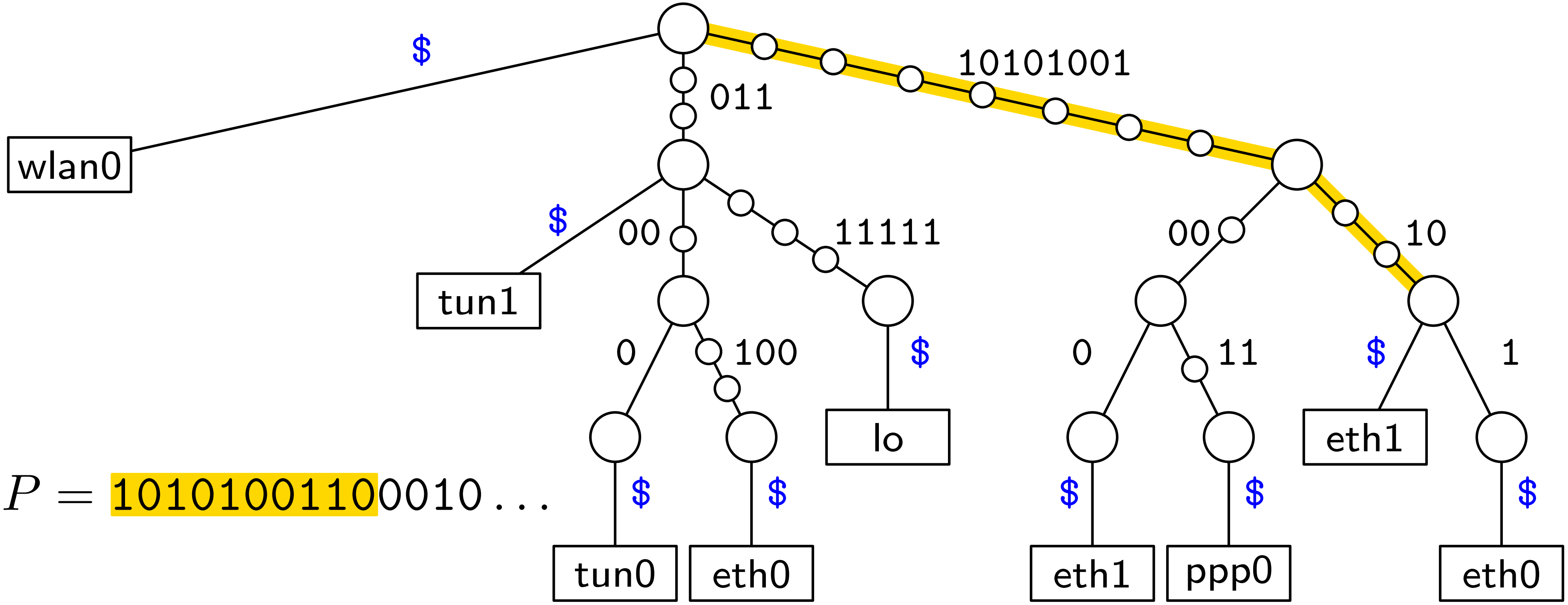
Build a trie T with all the addresses in the routing table.



- Find the node v corresponding to the maximal prefix that matches P
- Walk up the tree searching for the deepest ancestor u of v incident to a "$" edge towards a leaf ℓ
- Route the packet towards the interface stored in ℓ

Application: Packet Routing

Build a trie T with all the addresses in the routing table.

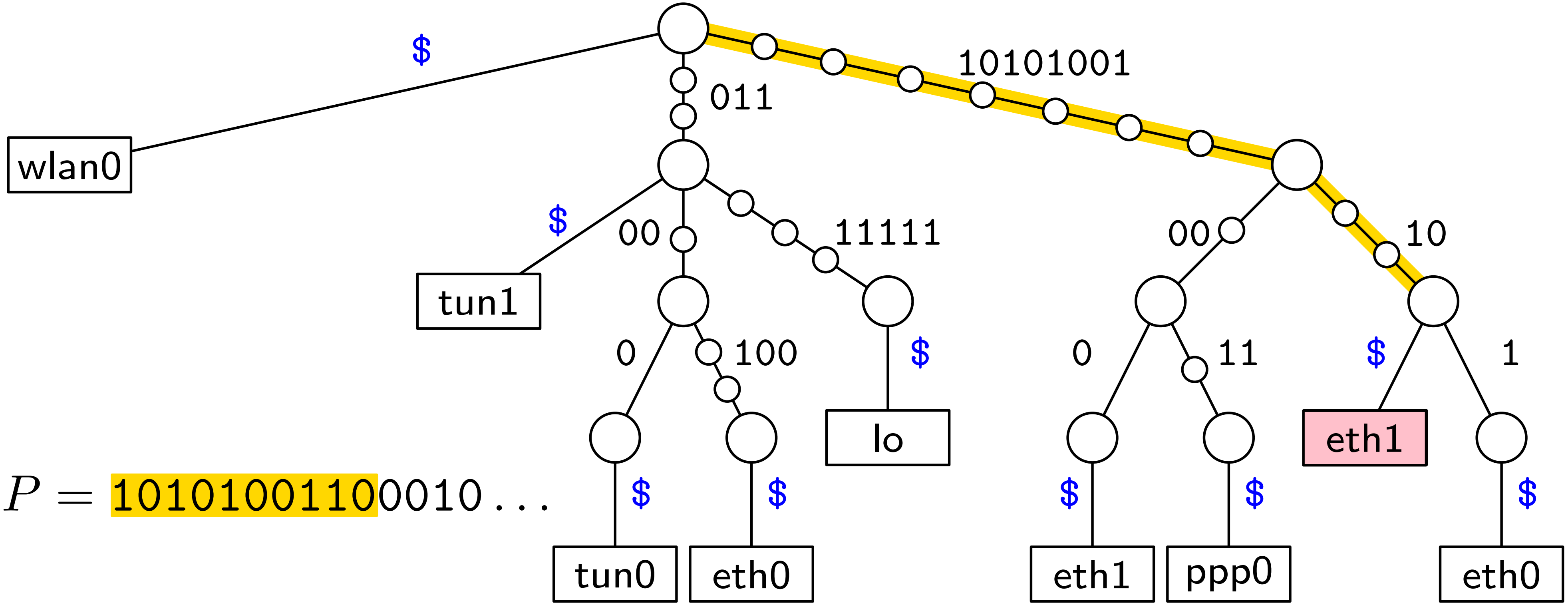


$P = 1010100110010\ldots$

- Find the node v corresponding to the maximal prefix that matches P
- Walk up the tree searching for the deepest ancestor u of v incident to a "$" edge towards a leaf ℓ
- Route the packet towards the interface stored in ℓ

Application: Packet Routing

Build a trie T with all the addresses in the routing table.

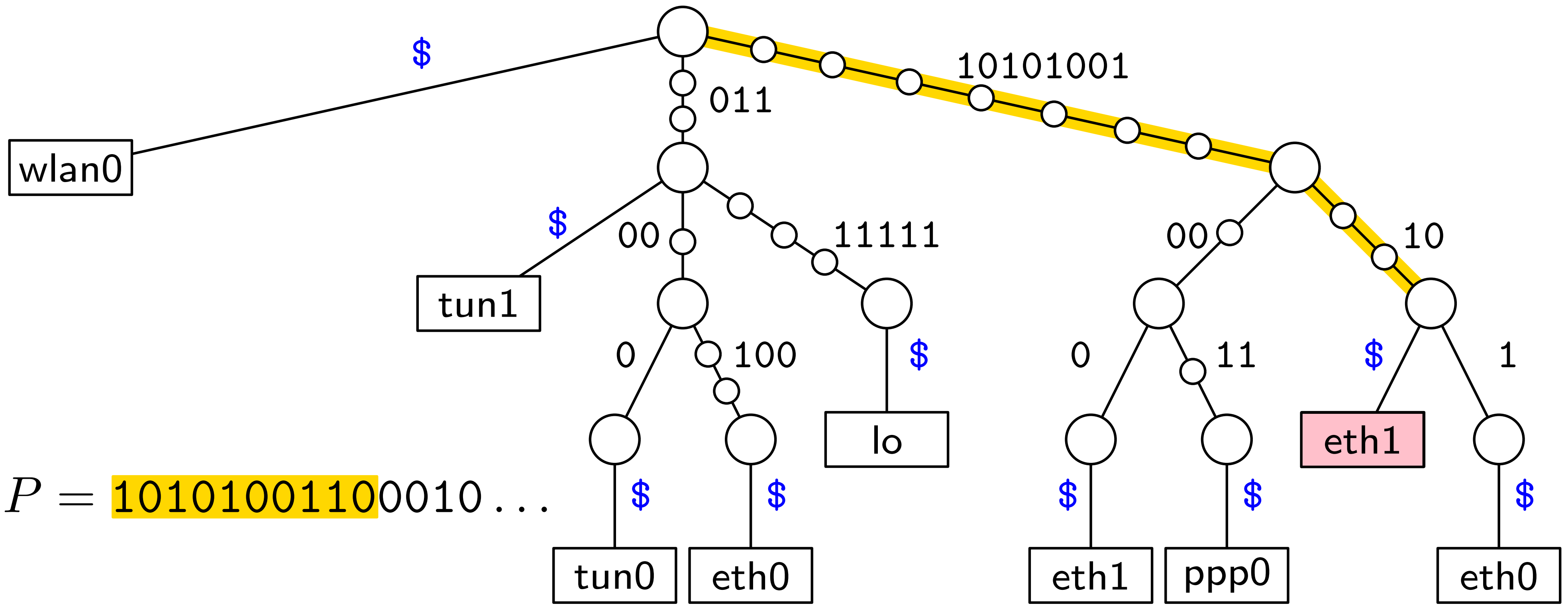


$P = 1010100110010\ldots$

- Find the node v corresponding to the maximal prefix that matches P
- Walk up the tree searching for the deepest ancestor u of v incident to a "$" edge towards a leaf ℓ
- Route the packet towards the interface stored in ℓ

Application: Packet Routing

Build a trie T with all the addresses in the routing table.



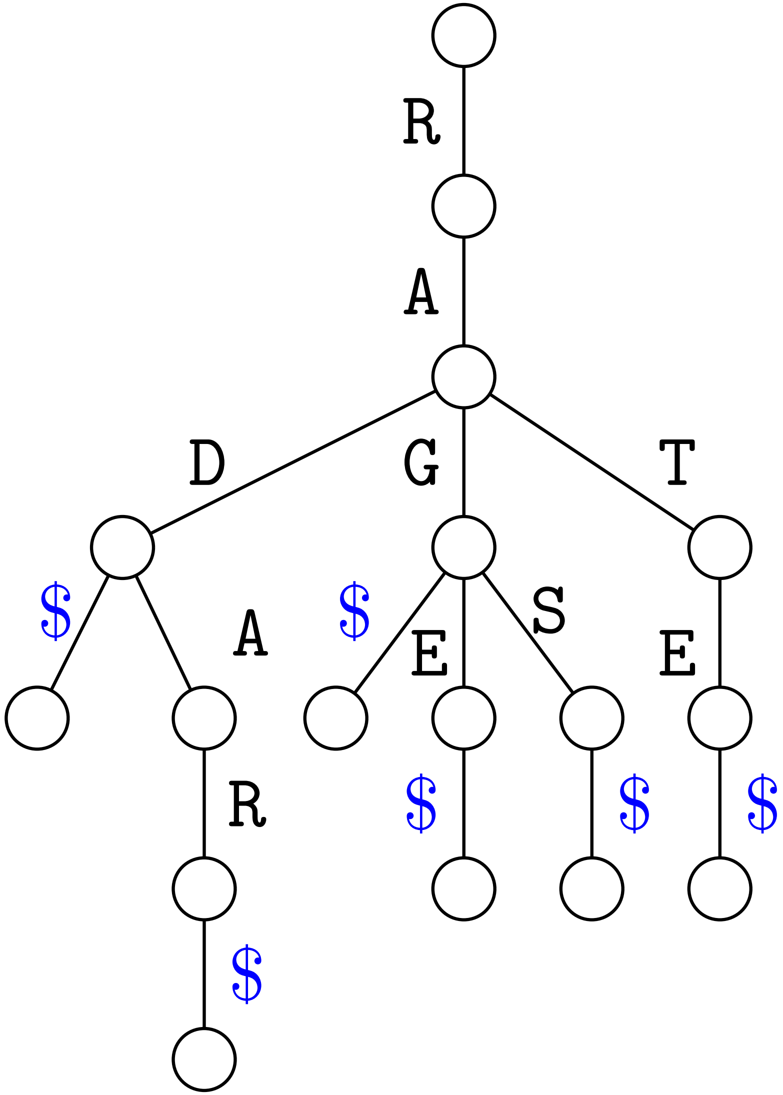
$P = 1010100110010\ldots$

- Find the node v corresponding to the maximal prefix that matches P
- Walk up the tree searching for the deepest ancestor u of v incident to a "$" edge towards a leaf ℓ
- Route the packet towards the interface stored in ℓ

Time: O(address length)

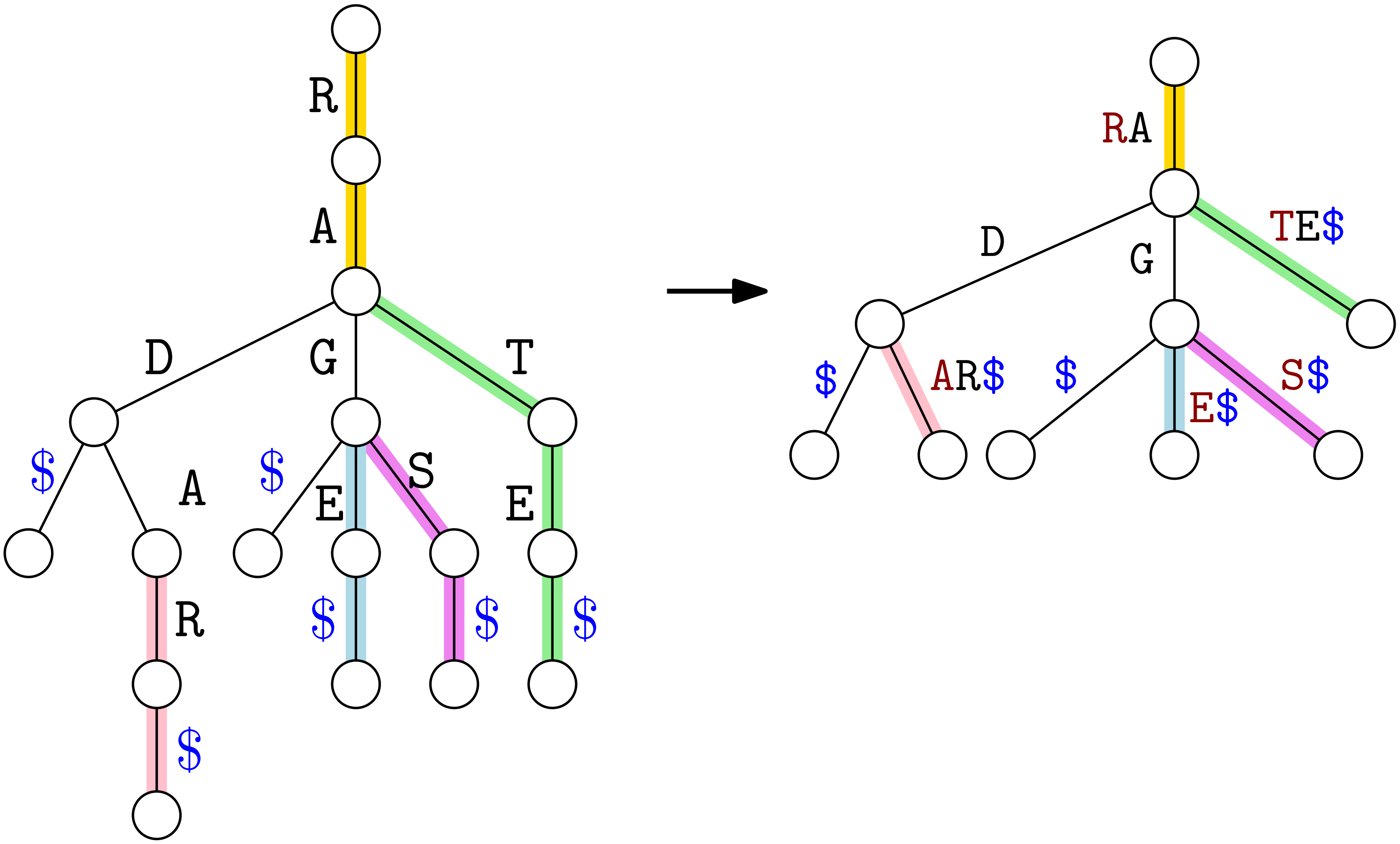
Compressed Tries (Radix Trees)

Contract non-branching paths to a single edge labelled with the corresponding substring



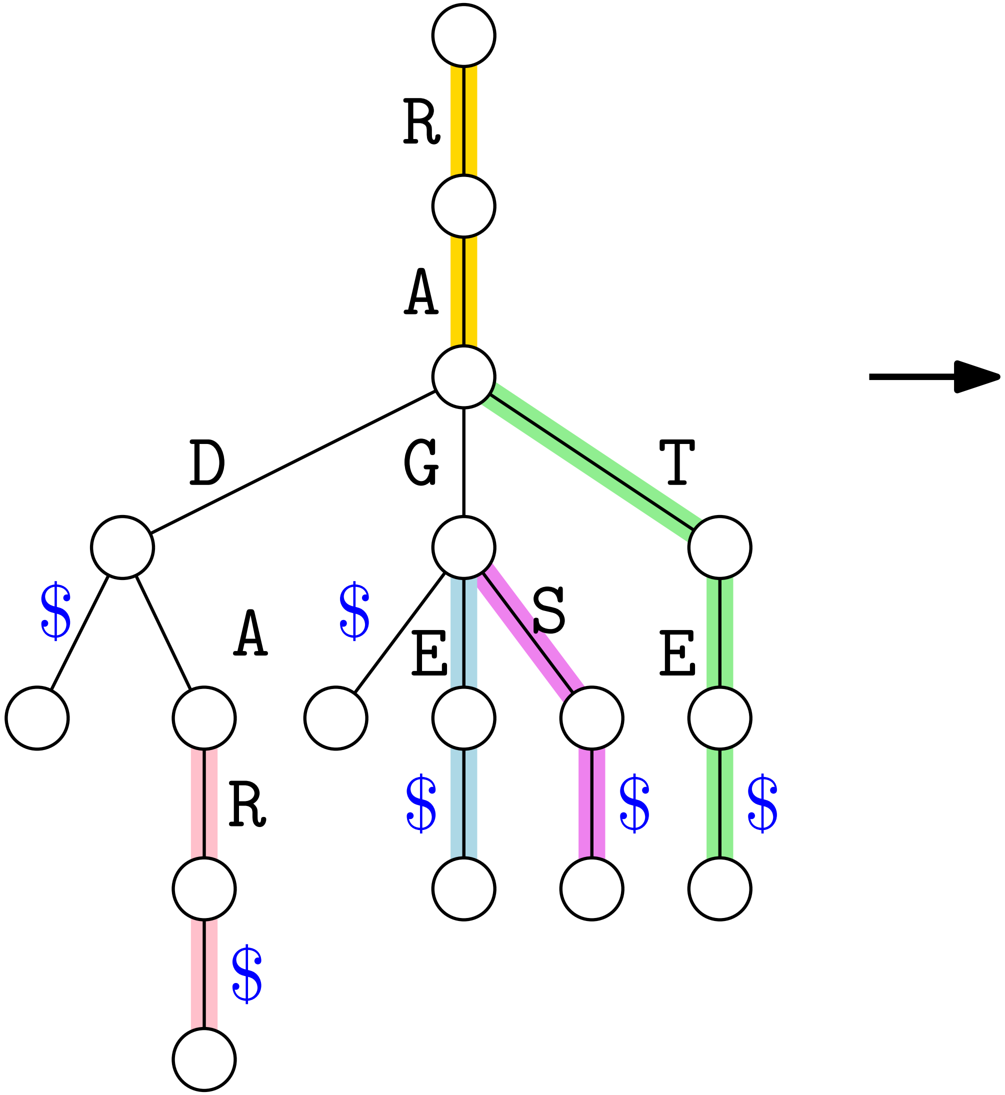
Compressed Tries (Radix Trees)

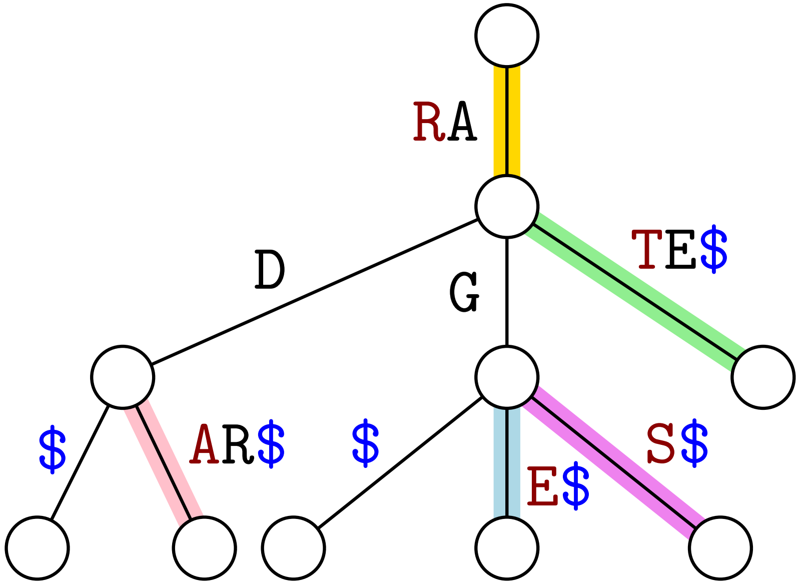
Contract non-branching paths to a single edge labelled with the corresponding substring



Compressed Tries (Radix Trees)

Contract non-branching paths to a single edge labelled with the corresponding substring



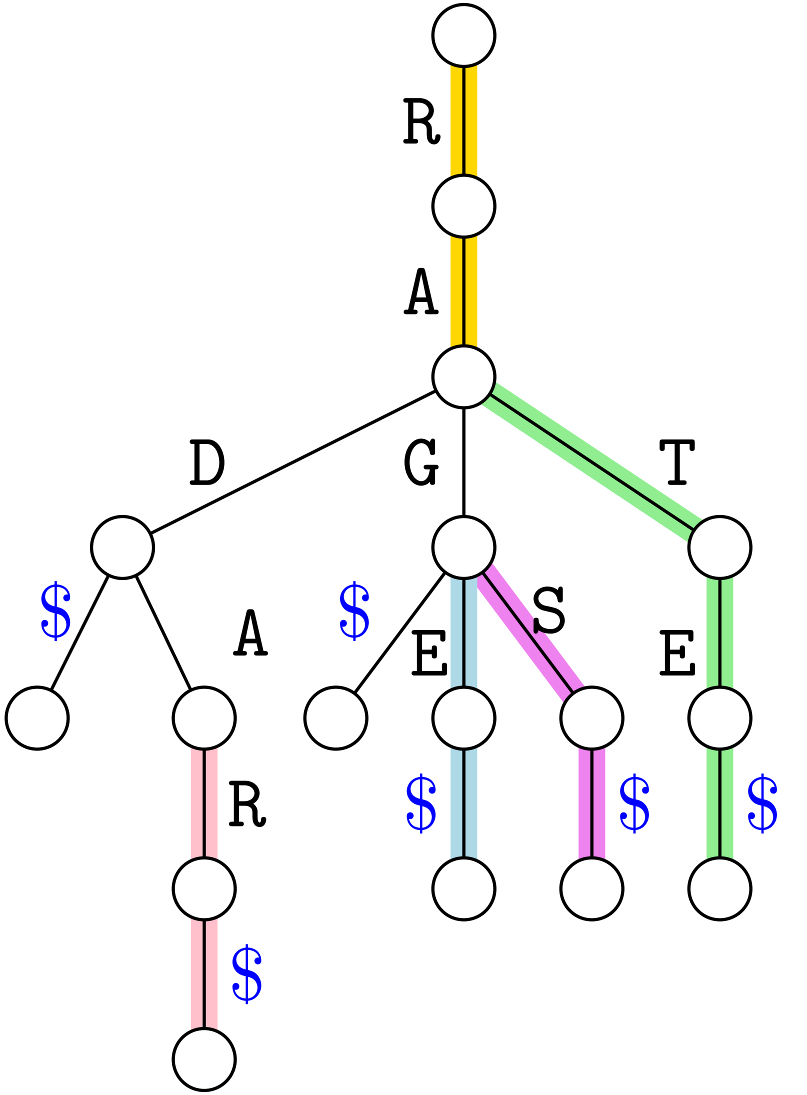


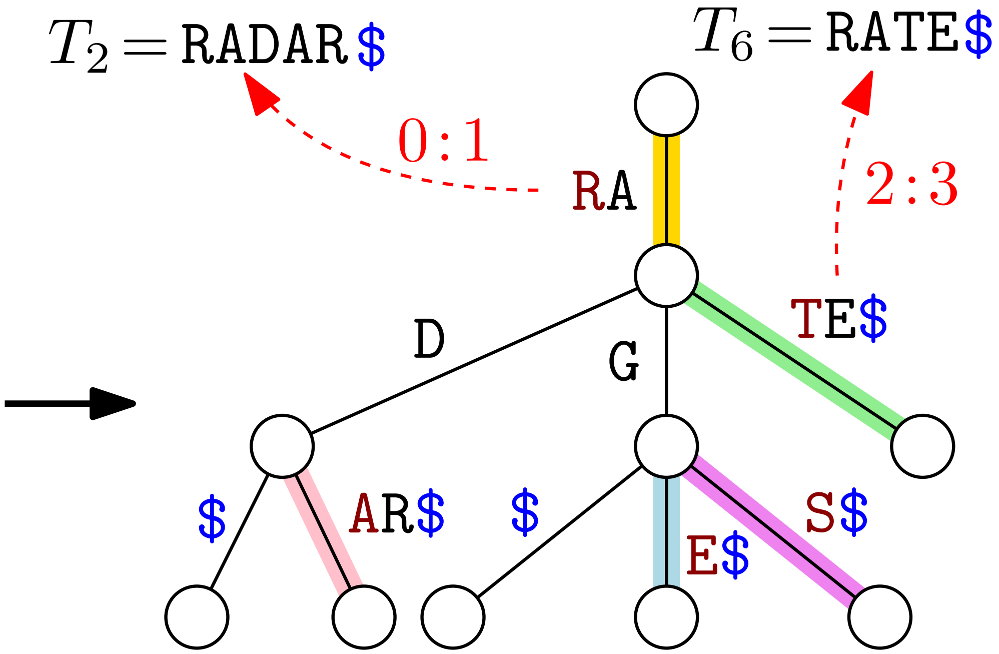
Previous constructions apply

Use the first character on each edge as the key

Compressed Tries (Radix Trees)

Contract non-branching paths to a single edge labelled with the corresponding substring





Previous constructions apply

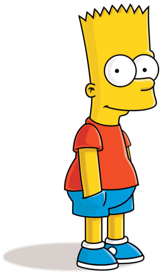
Use the first character on each edge as the key

Store edge labels as indices in the input strings

Suffix Trees

Back to String Matching

Problem: Given an alphabet Σ, a *text* $T \in \Sigma^*$ and a *pattern* $P \in \Sigma^*$, find some occurrence/all occurrences of P in T.



$$\Sigma = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}, \textvisiblespace\}$$

$$T = \texttt{B}\underline{\texttt{art}}\texttt{\textvisiblespace played\textvisiblespace d}\underline{\texttt{art}}\texttt{s\textvisiblespace at\textvisiblespace the\textvisiblespace p}\underline{\texttt{art}}\texttt{y}$$

$$P = \texttt{art}$$

Want: A data structure that can preprocesses T and answer string matching queries

Suffix Trees

The **suffix tree** of T is the compressed trie of all the suffixes of $T\$$

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{N}, \mathtt{S}\}$ $\qquad T = \mathtt{BANANAS}\$$

Suffix Trees

The **suffix tree** of T is the compressed trie of all the suffixes of $T\$$

$$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{N}, \mathtt{S}\} \qquad T = \overset{\mathtt{01234567}}{\mathtt{BANANAS\$}}$$

7	$\mathtt{\$}$
6	$\mathtt{S\$}$
5	$\mathtt{AS\$}$
4	$\mathtt{NAS\$}$
3	$\mathtt{ANAS\$}$
2	$\mathtt{NANAS\$}$
1	$\mathtt{ANANAS\$}$
0	$\mathtt{BANANAS\$}$

Suffix Trees

The **suffix tree** of T is the compressed trie of all the suffixes of $T\$$

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{N}, \mathtt{S}\}$ $\quad$ $T = $ `BANANAS$` (positions 01234567)

7 $

6 S$

5 AS$

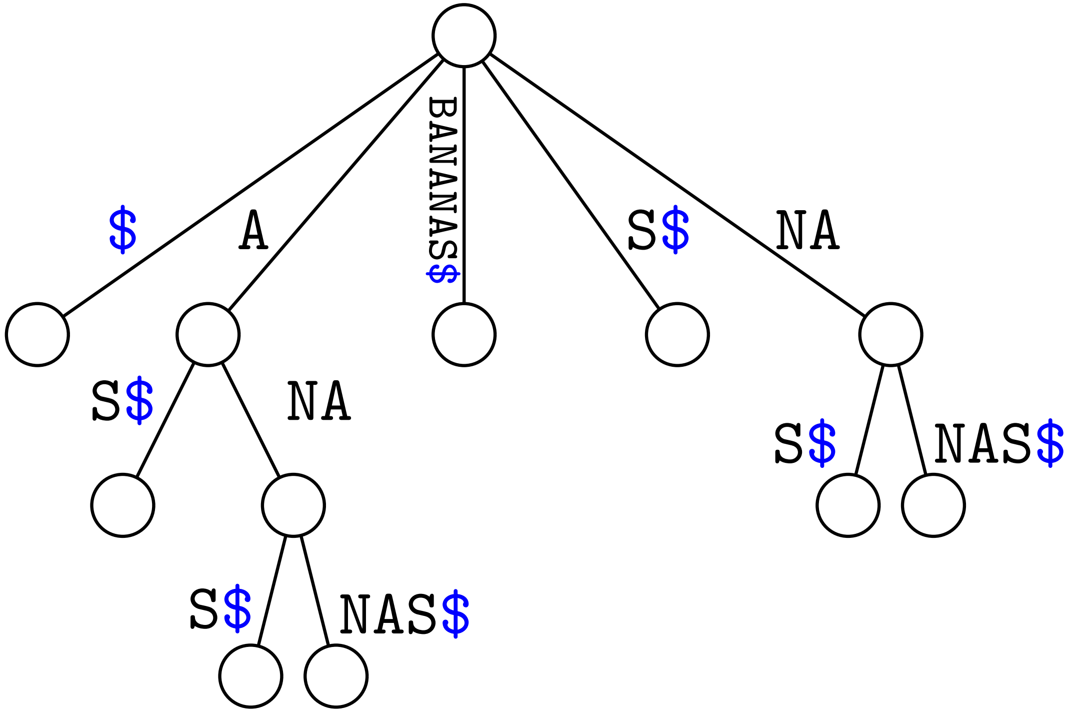
4 NAS$

3 ANAS$

2 NANAS$

1 ANANAS$

0 BANANAS$

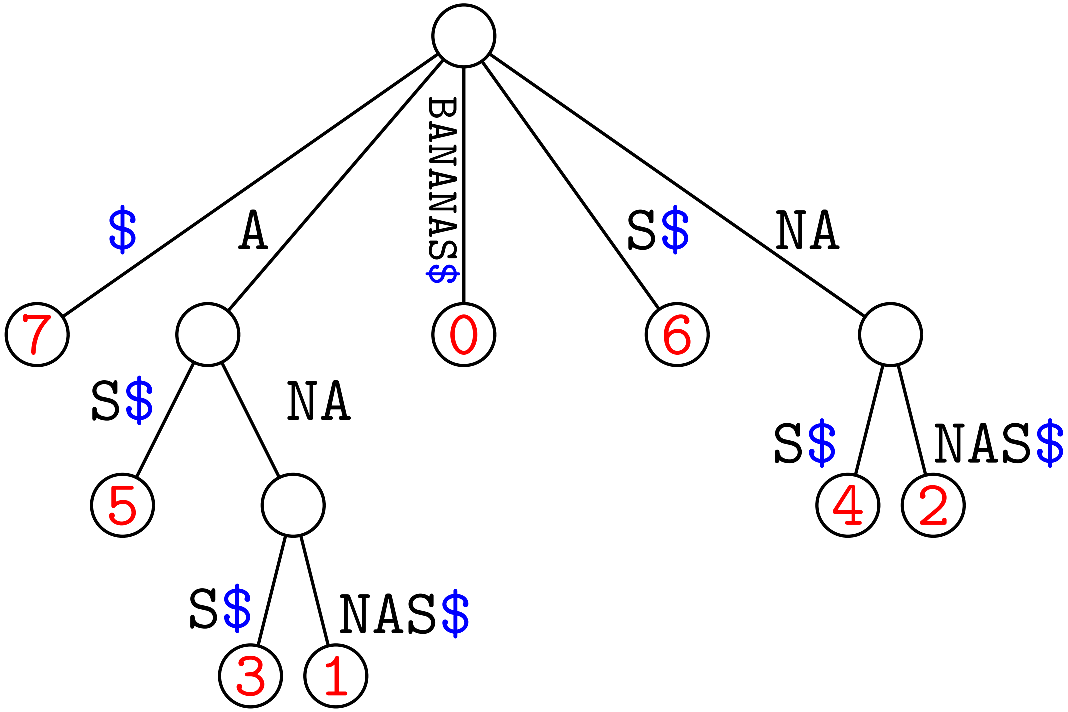


Suffix Trees

The **suffix tree** of T is the compressed trie of all the suffixes of $T\$$

$$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{N}, \mathtt{S}\} \qquad T = \overset{\mathtt{01234567}}{\mathtt{BANANAS\$}}$$

7	\$
6	S\$
5	AS\$
4	NAS\$
3	ANAS\$
2	NANAS\$
1	ANANAS\$
0	BANANAS\$



Label edges with indices into T

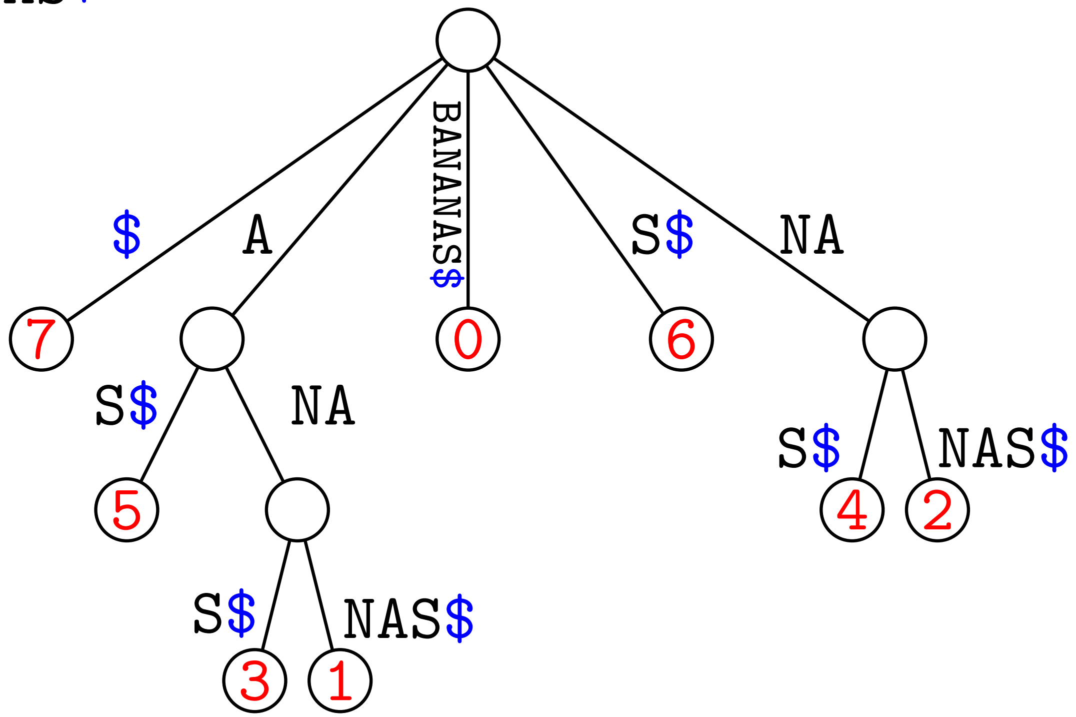
Label leaves with the index of the start of the corresponding suffix

Suffix Trees

The **suffix tree** of T is the compressed trie of all the suffixes of $T\$$

$\Sigma = \{\texttt{A}, \texttt{B}, \texttt{N}, \texttt{S}\}$ $\qquad T = \overset{\texttt{01234567}}{\texttt{BANANAS\$}}$

7	$
6	S$
5	AS$
4	NAS$
3	ANAS$
2	NANAS$
1	ANANAS$
0	BANANAS$

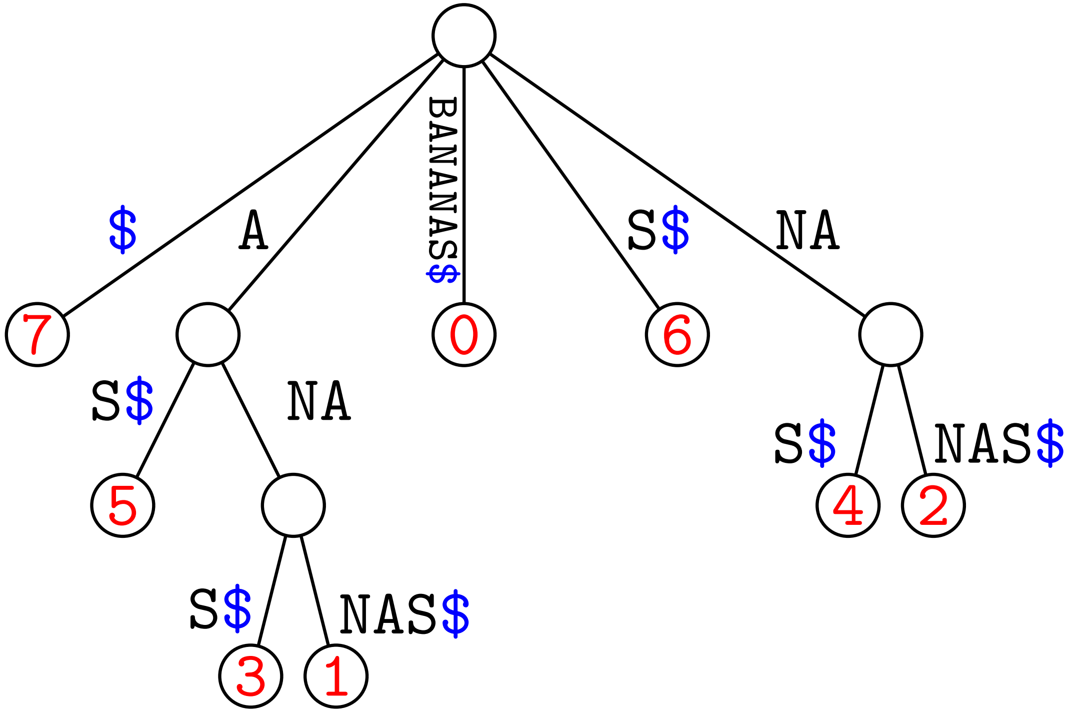


Label edges with indices into T

Label leaves with the index of the start of the corresponding suffix

Space: $\quad O(\#\text{ nodes}) = O(\#\text{ leaves}) = O(|T|)$

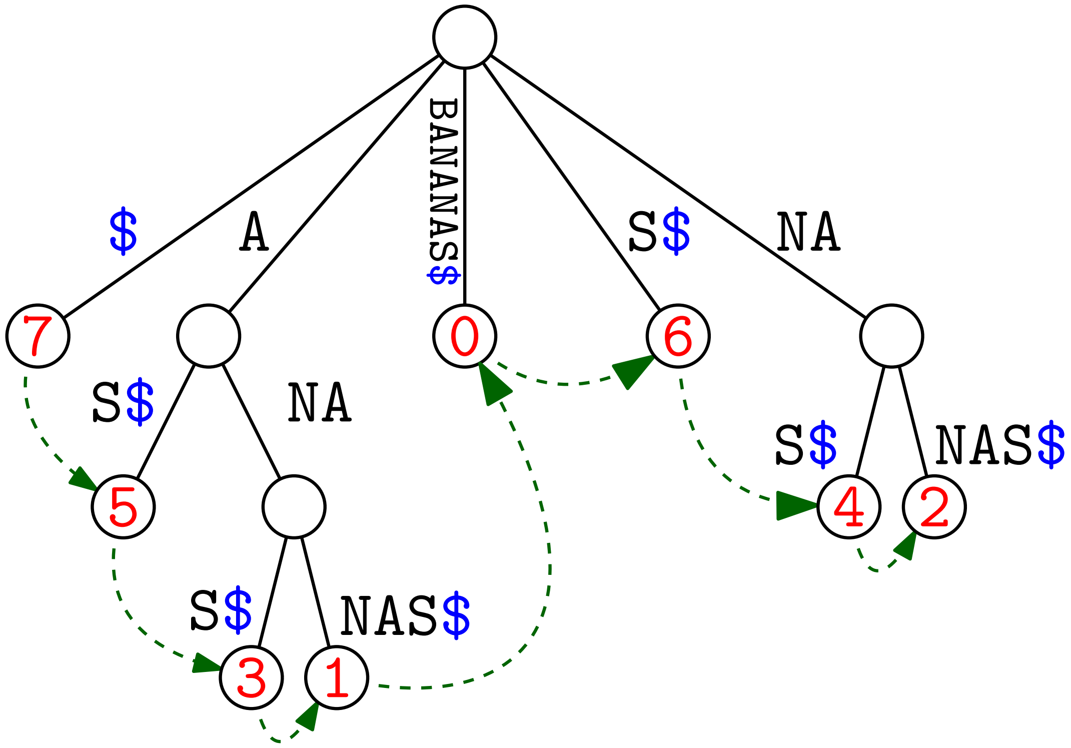
Applications: String Matching



Searching for a pattern P returns a compact representation of **all** occurrences of P in T

- Find the node v corresponding to P
- The occurrences of P are all and only the leaves in the subtree of v

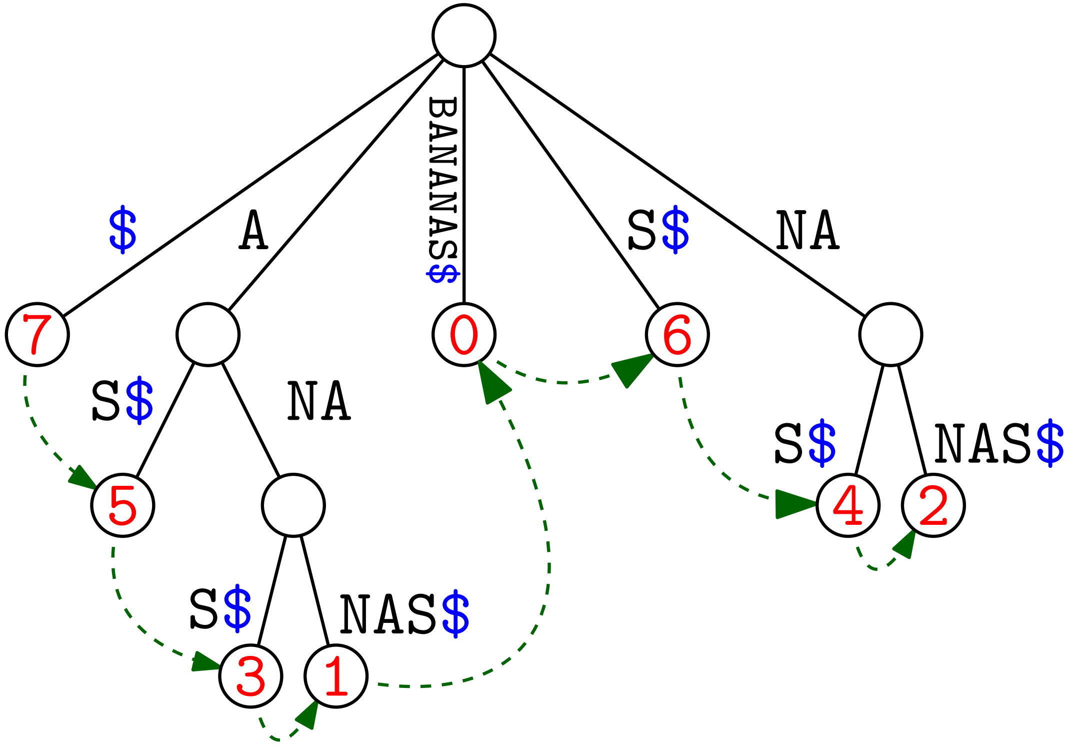
Applications: String Matching



Searching for a pattern P returns a compact representation of **all** occurrences of P in T

- Find the node v corresponding to P
- The occurrences of P are all and only the leaves in the subtree of v
- Arrange leaves in a linked list to find the next match in $O(1)$ time

Applications: String Matching

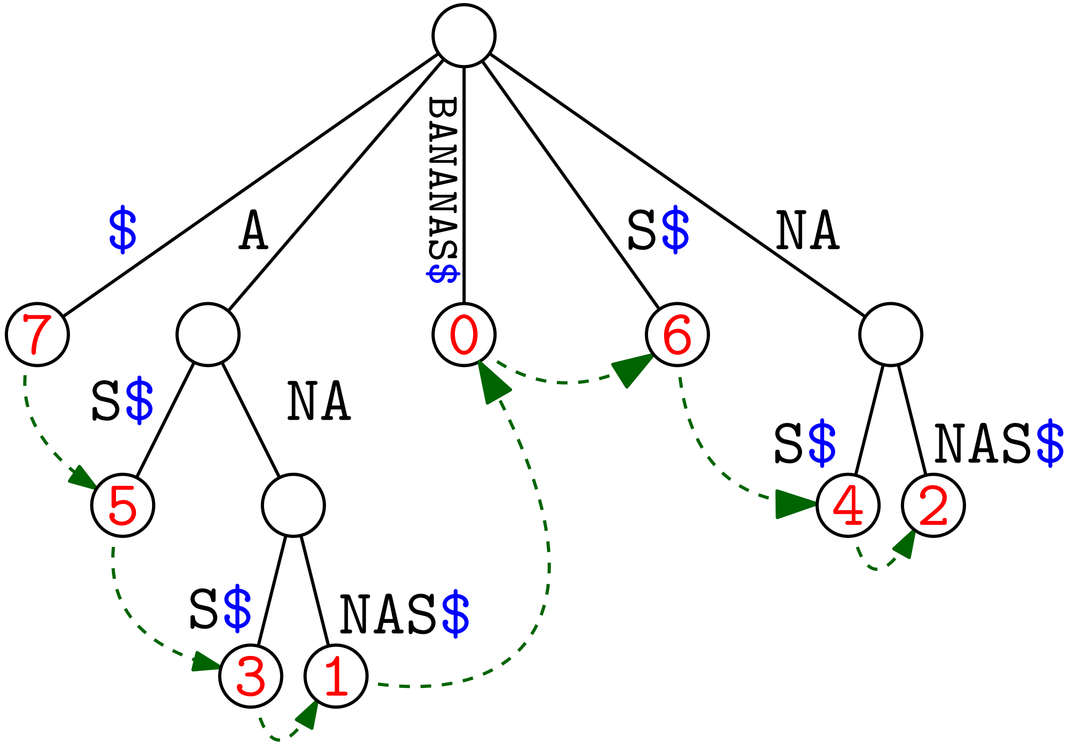


Searching for a pattern P returns a compact representation of **all** occurrences of P in T

- Find the node v corresponding to P
- The occurrences of P are all and only the leaves in the subtree of v
- Arrange leaves in a linked list to find the next match in $O(1)$ time

Time: $O(|P| + \log|\Sigma| + \#\text{desired matches})$

Applications: String Matching



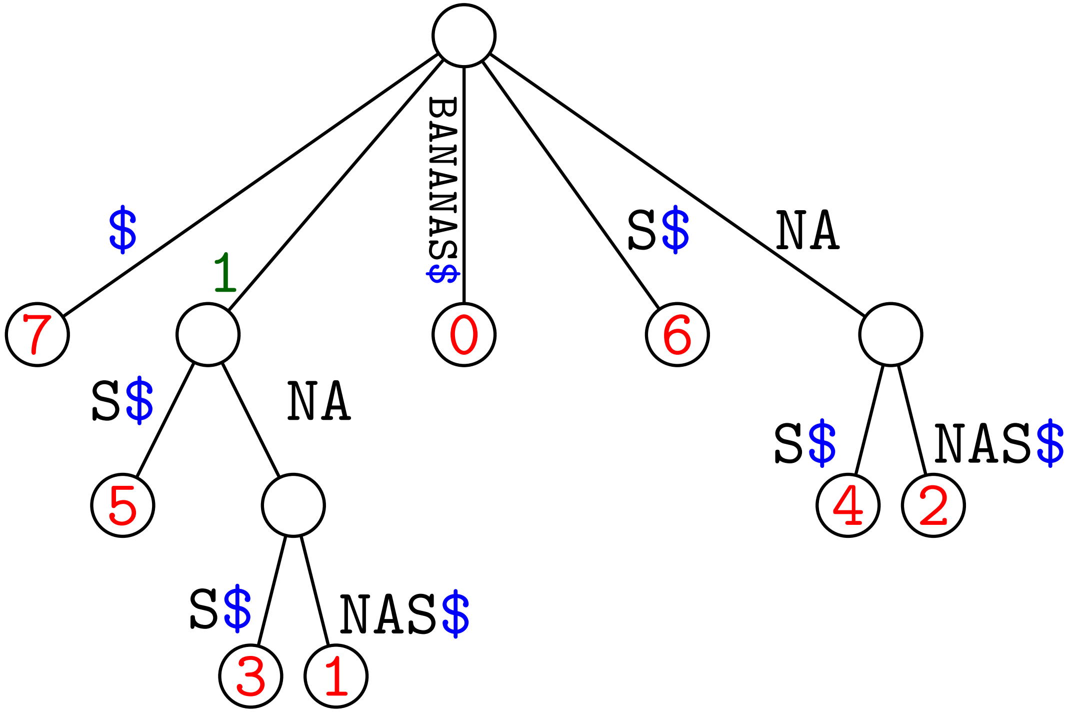
Searching for a pattern P returns a compact representation of **all** occurrences of P in T

- Find the node v corresponding to P
- The occurrences of P are all and only the leaves in the subtree of v
- Arrange leaves in a linked list to find the next match in $O(1)$ time

Time: $O(|P| + \log|\Sigma| + \#\text{desired matches})$

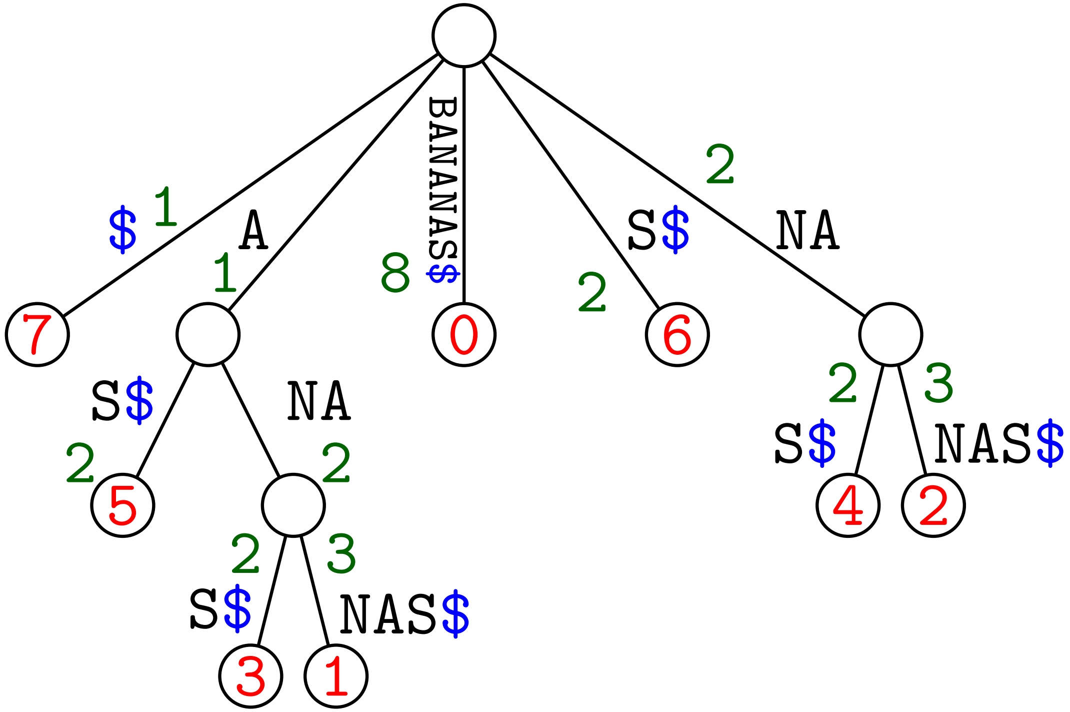
Number of matches in time $O(|P| + \log|\Sigma|)$ (store # leaves in the subtree)

Applications: Longest Repeated Substring



Find the longest string that appears at least twice in T as a substring:

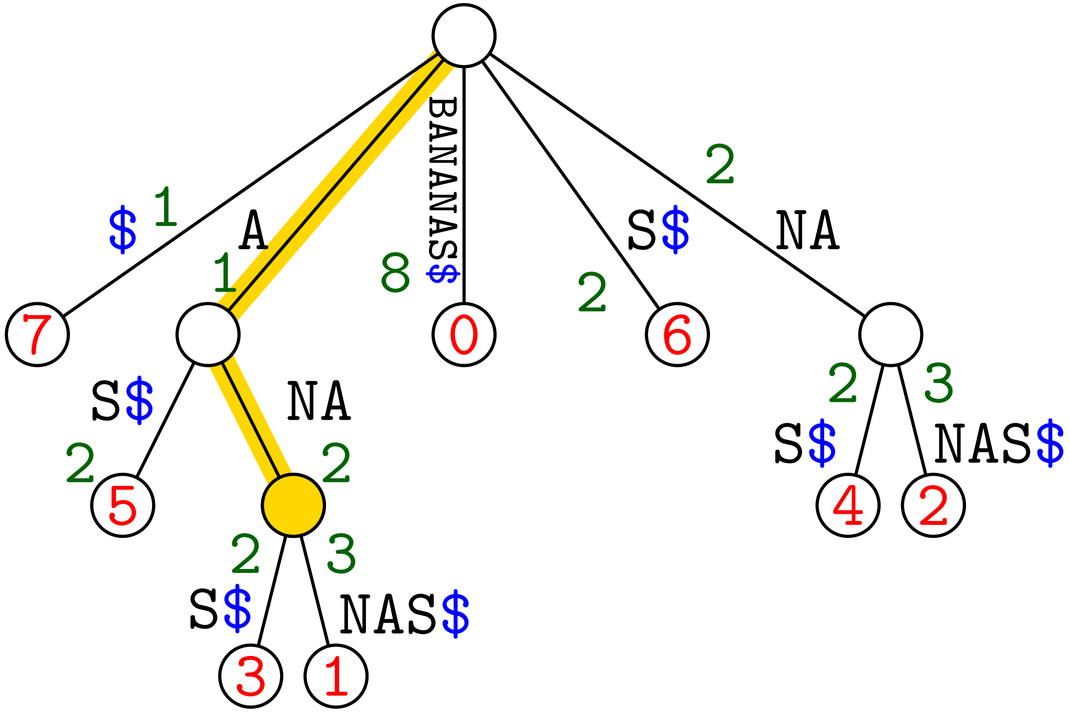
Applications: Longest Repeated Substring



Find the longest string that appears at least twice in T as a substring:

- Assign a length to each edge equal to the number of symbols in its label

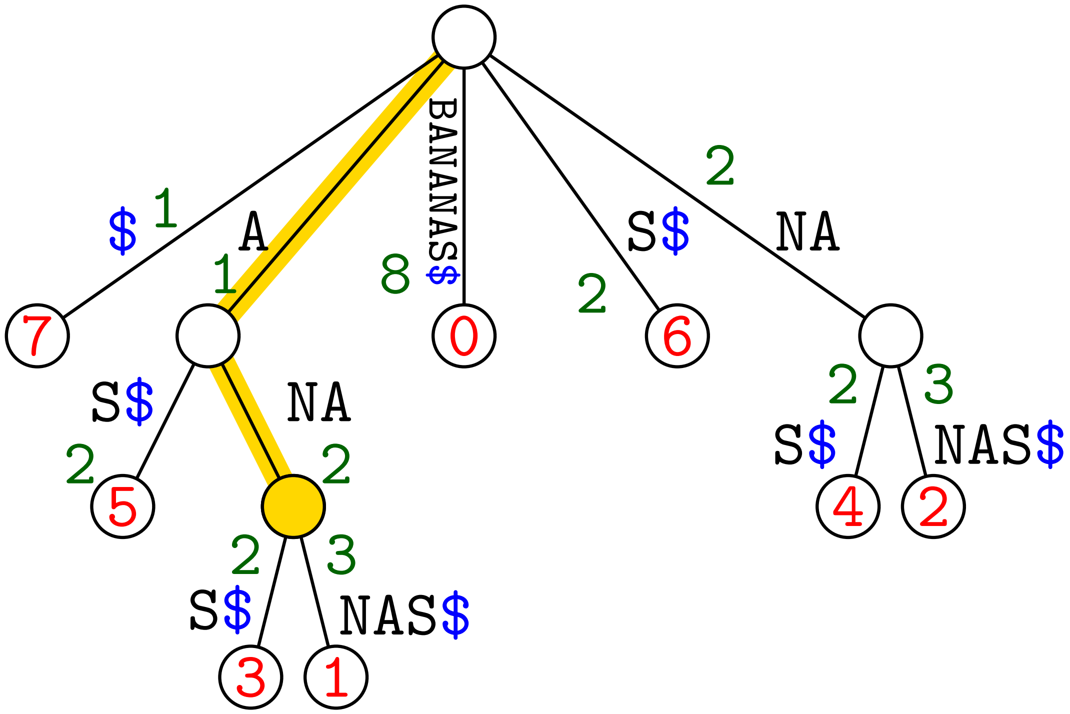
Applications: Longest Repeated Substring



Find the longest string that appears at least twice in T as a substring:

- Assign a length to each edge equal to the number of symbols in its label
- Find the deepest (w.r.t. edge lengths) node with at least two descendants

Applications: Longest Repeated Substring

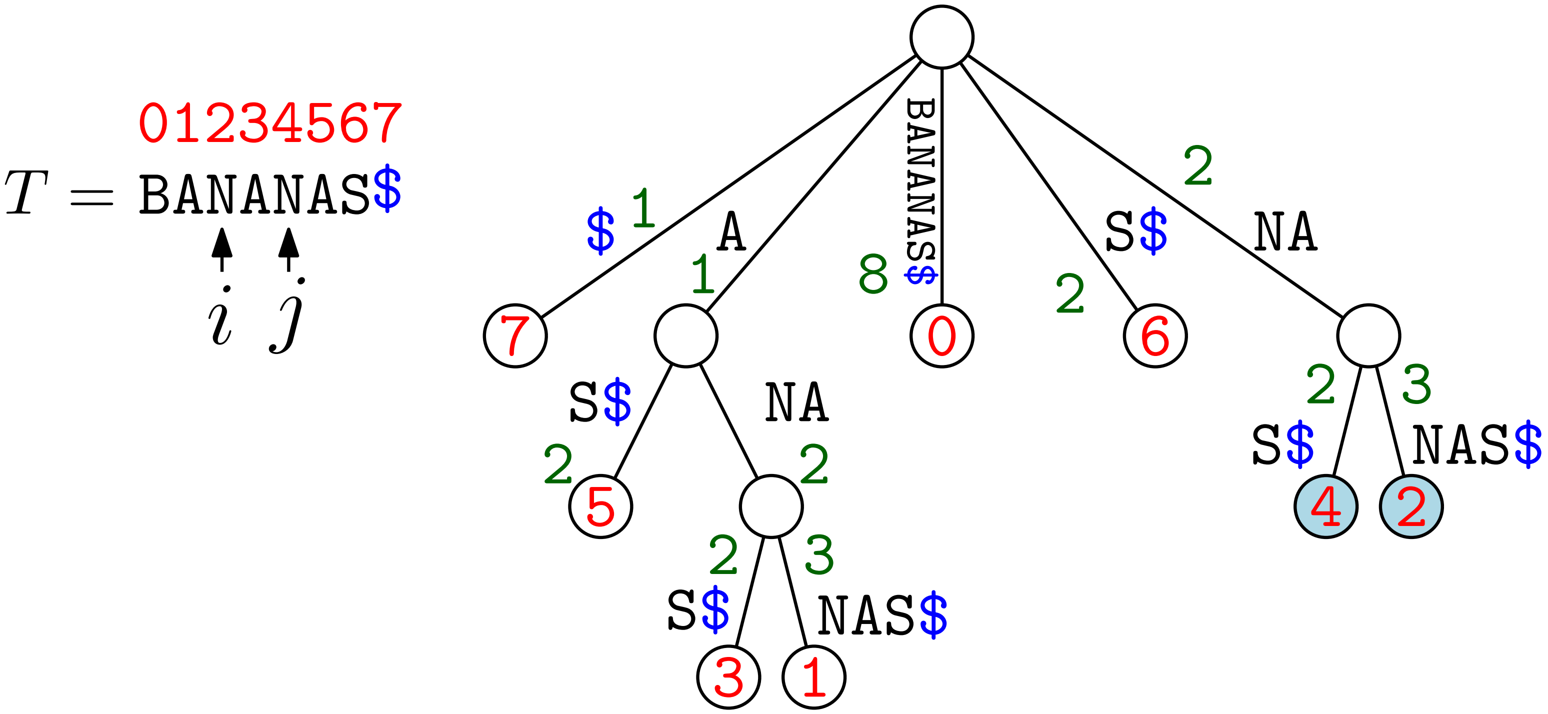


Find the longest string that appears at least twice in T as a substring:

- Assign a length to each edge equal to the number of symbols in its label
- Find the deepest (w.r.t. edge lengths) node with at least two descendants

Time: $O(|T|)$

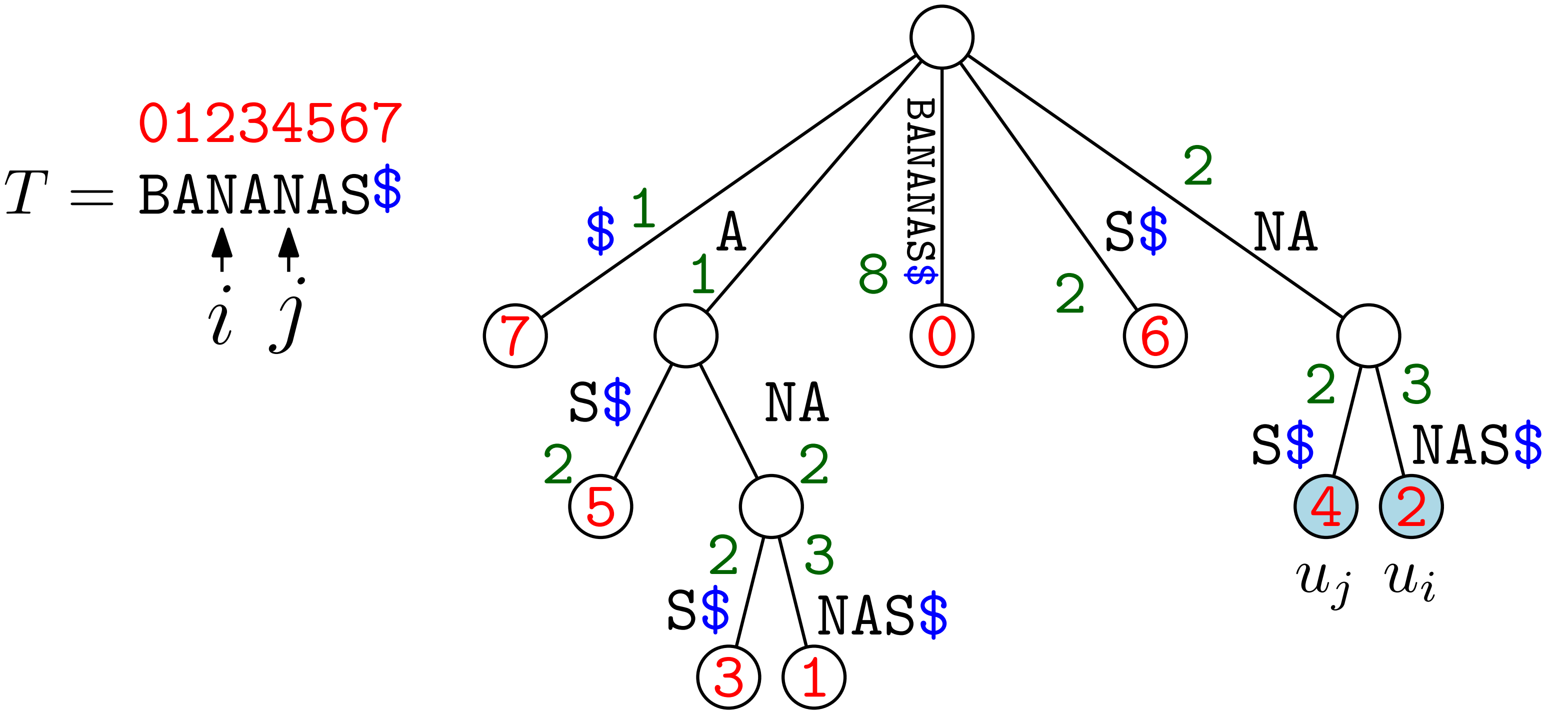
Applications: Longest Common Prefix



Given indicies i and j, find the longest common prefix of $T[i:]$ and $T[j:]$

- Look at the leaves u_i, u_j corresponding to $T[i:]$ and $T[j:]$

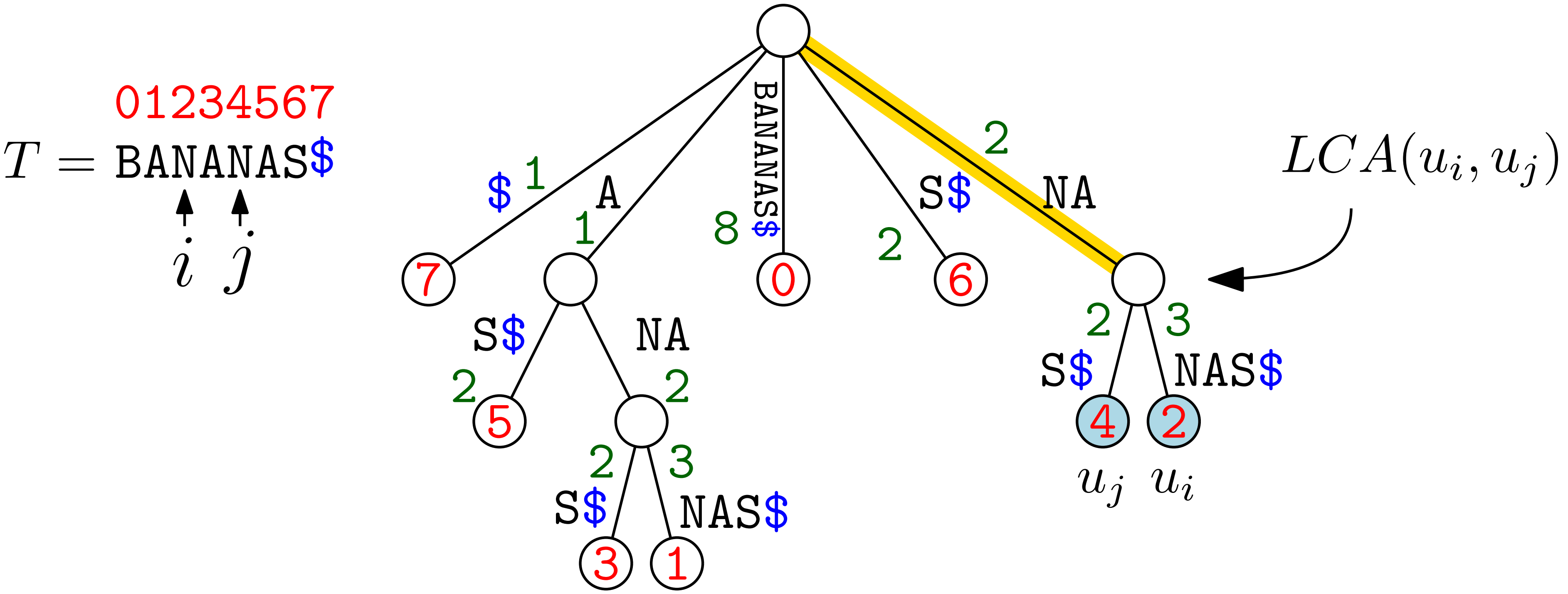
Applications: Longest Common Prefix



Given indicies i and j, find the longest common prefix of $T[i:]$ and $T[j:]$

- Look at the leaves u_i, u_j corresponding to $T[i:]$ and $T[j:]$
- Find the common prefix of the paths from the root to u_i and u_j

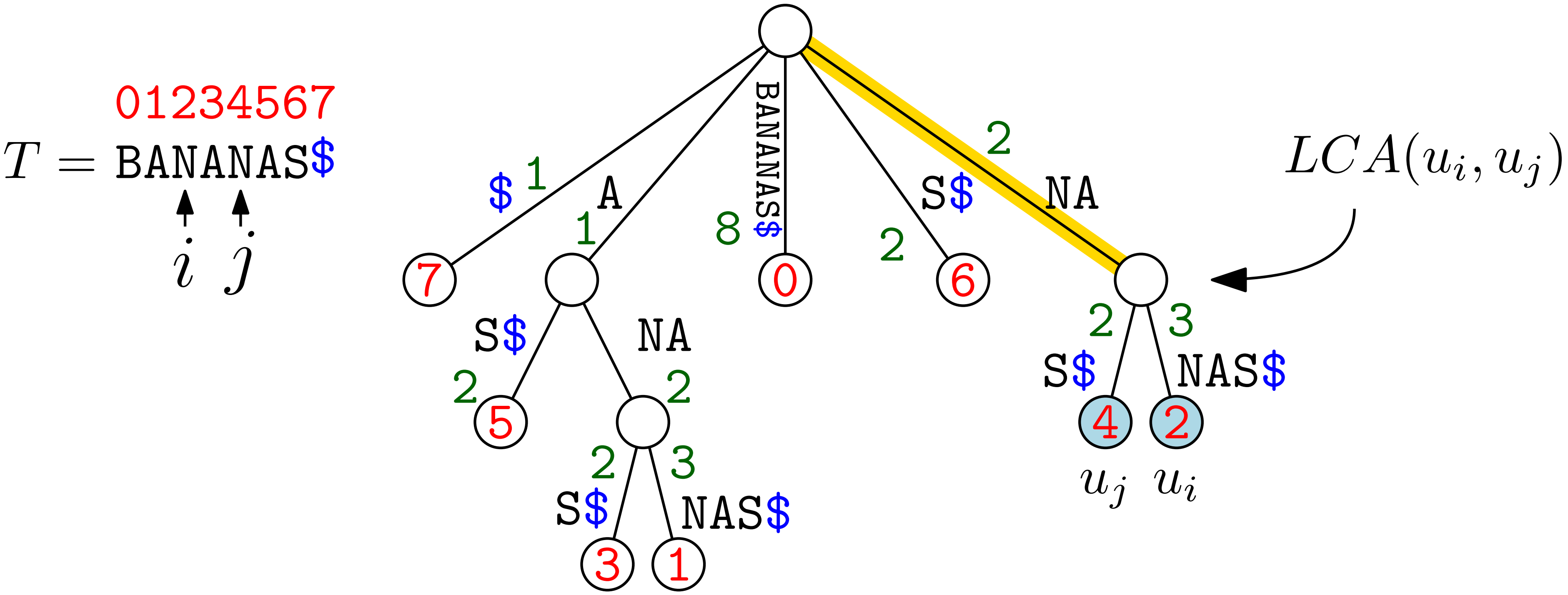
Applications: Longest Common Prefix



Given indicies i and j, find the longest common prefix of $T[i:]$ and $T[j:]$

- Look at the leaves u_i, u_j corresponding to $T[i:]$ and $T[j:]$
- Find the common prefix of the paths from the root to u_i and u_j
- This is the path from the root to the lowest common ancestor of u_i and u_j

Applications: Longest Common Prefix



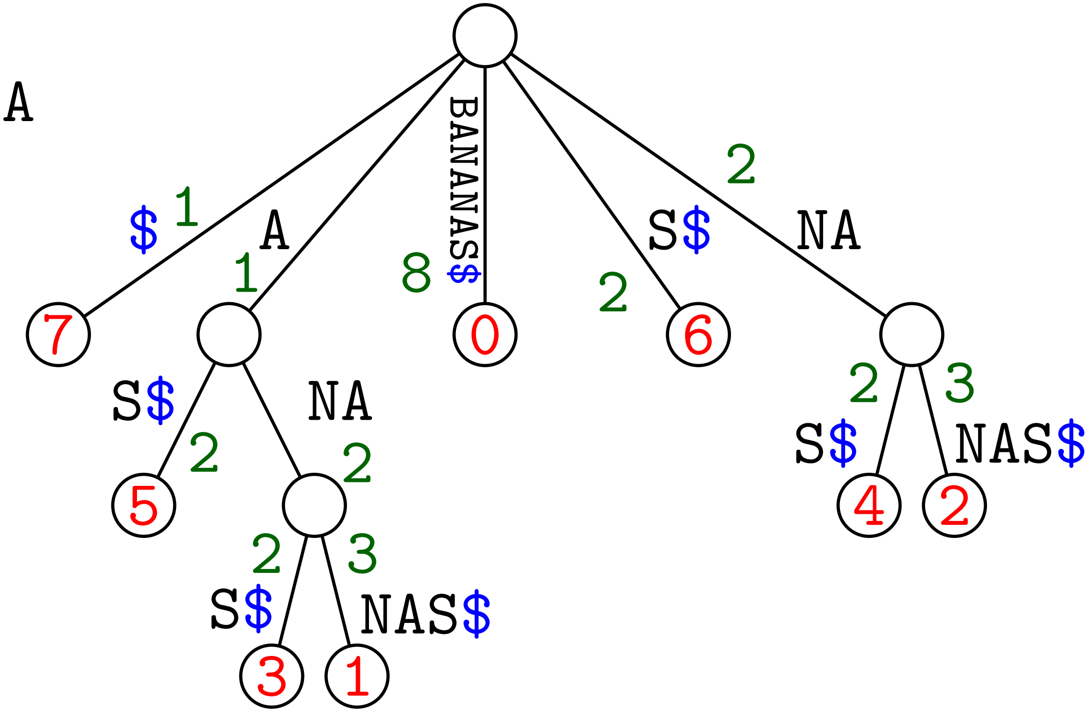
Given indicies i and j, find the longest common prefix of $T[i:]$ and $T[j:]$

- Look at the leaves u_i, u_j corresponding to $T[i:]$ and $T[j:]$
- Find the common prefix of the paths from the root to u_i and u_j
- This is the path from the root to the lowest common ancestor of u_i and u_j

We already know how to answer LCA queries in constant time!

Applications: Fiding Additional Matches

$P = T[2:3] =$ NA

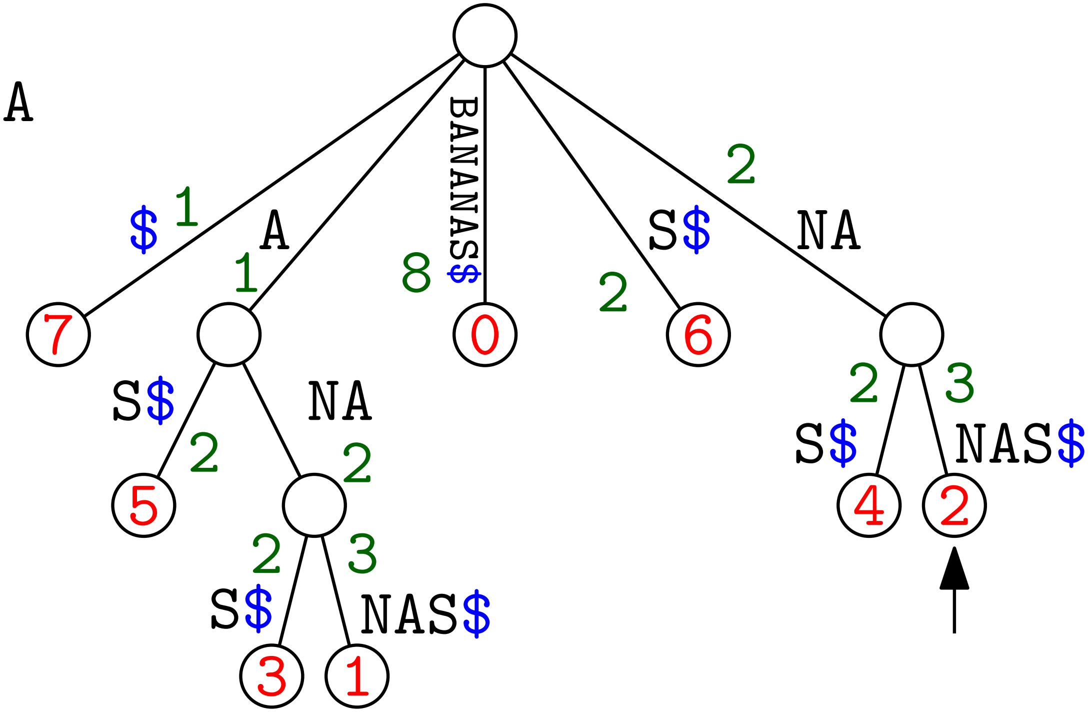


Given an occurrence $T[i:j]$ of P in T, find all other occurrences of P:

- We want to quickly find the node that corresponds to P

Applications: Fiding Additional Matches

$P = T[2:3] =$ NA

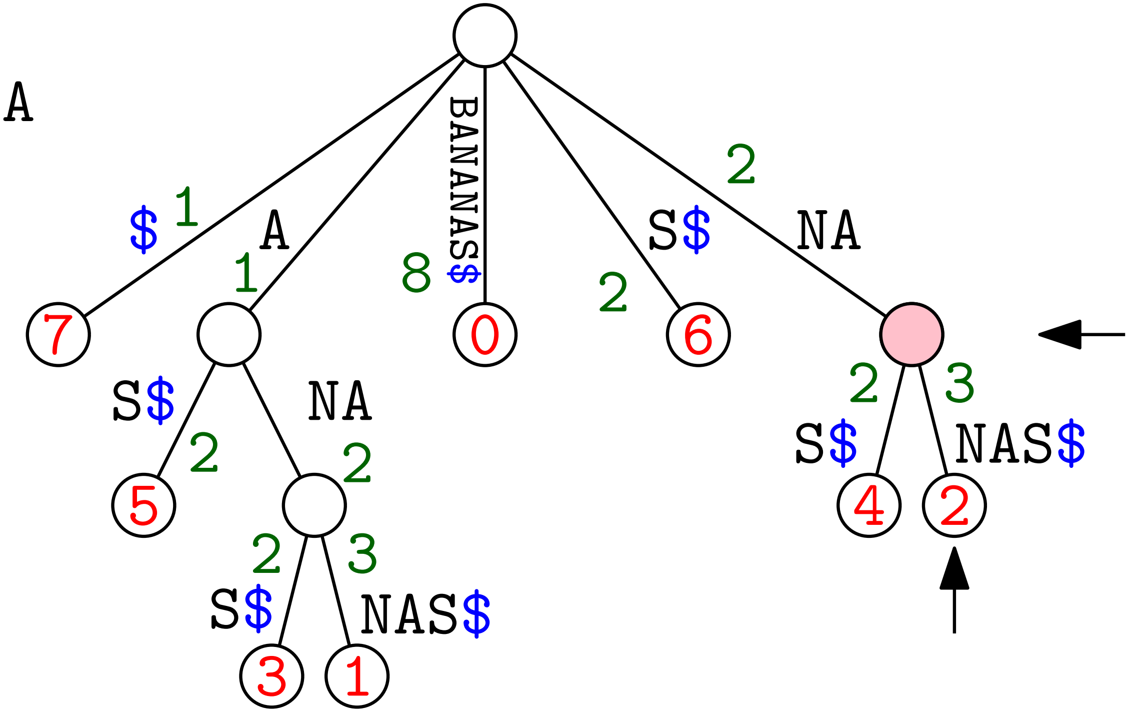


Given an occurrence $T[i:j]$ of P in T, find all other occurrences of P:

- We want to quickly find the node that corresponds to P
- Start from the leaf corresponding to $T[i:]$

Applications: Fiding Additional Matches

$P = T[2:3] =$ NA

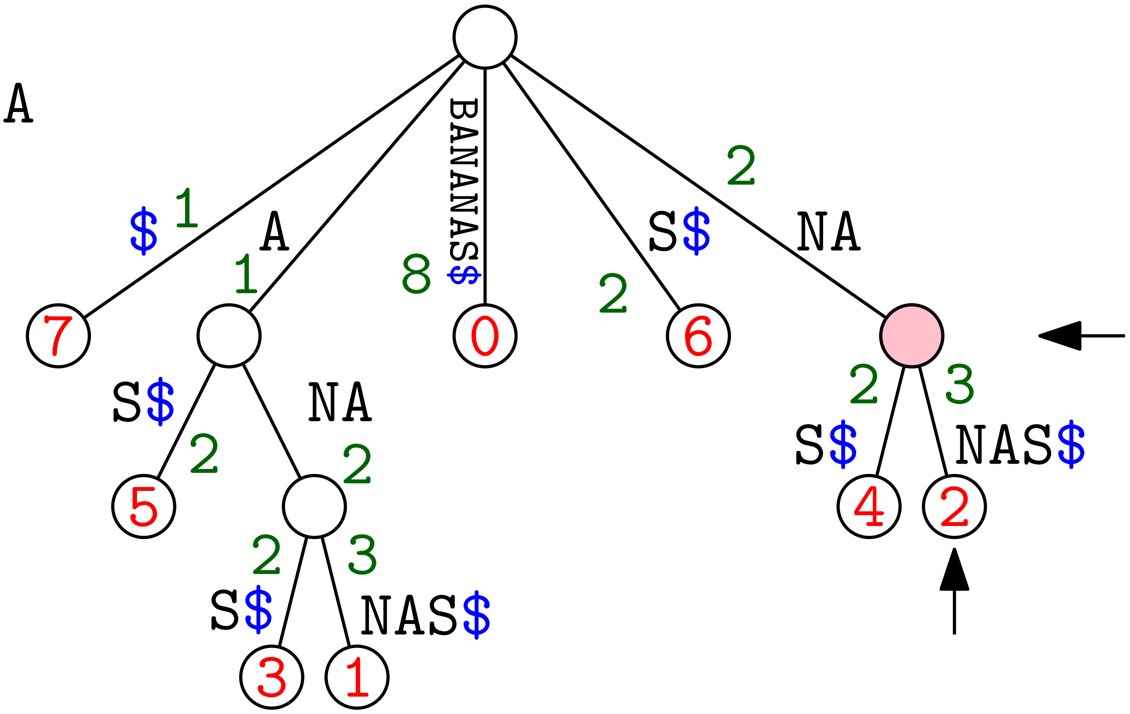


Given an occurrence $T[i:j]$ of P in T, find all other occurrences of P:

- We want to quickly find the node that corresponds to P
- Start from the leaf corresponding to $T[i:]$
- Walk **up** the tree for "$|T| - j$" characters

Applications: Fiding Additional Matches

$P = T[2:3] = $ NA

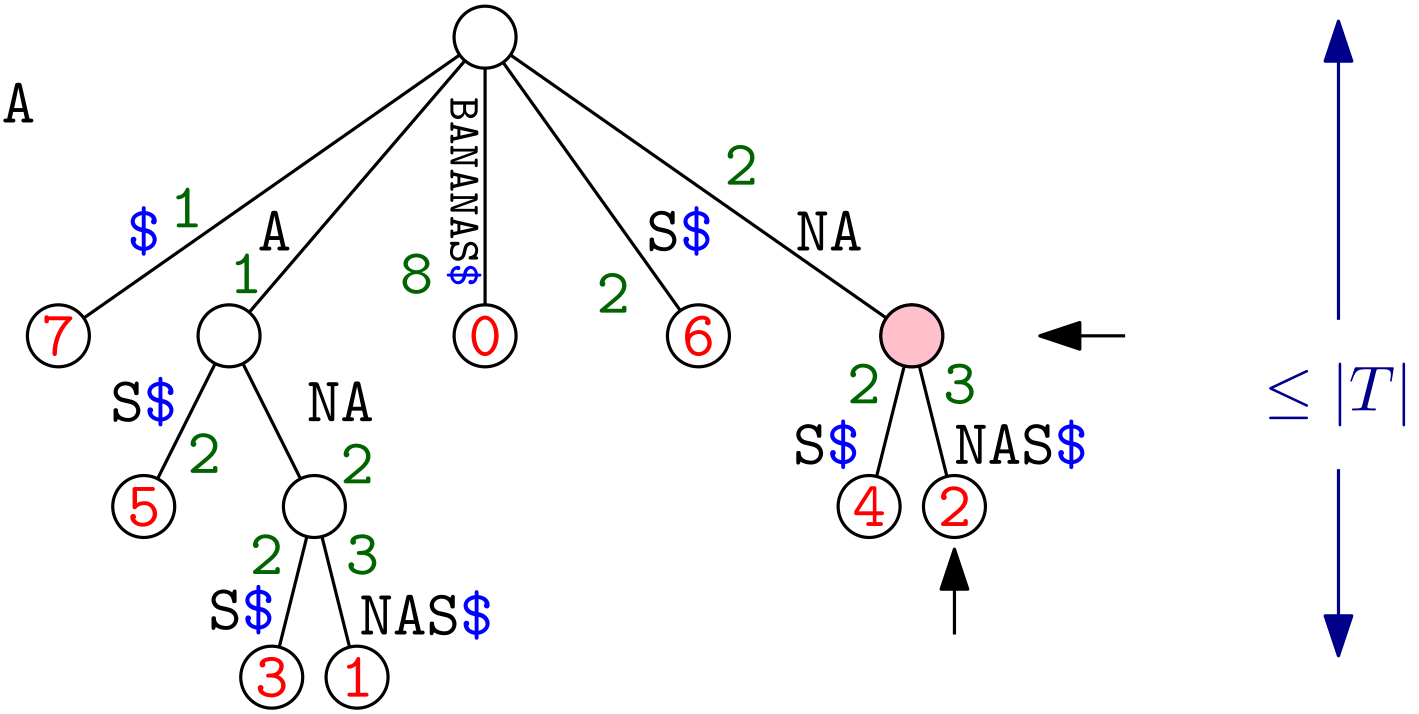


Given an occurrence $T[i:j]$ of P in T, find all other occurrences of P:

- We want to quickly find the node that corresponds to P
- Start from the leaf corresponding to $T[i:]$
- Walk **up** the tree for "$|T| - j$" characters
- This is a **weighted** level ancestor query!

Applications: Fiding Additional Matches

$P = T[2:3] = \texttt{NA}$



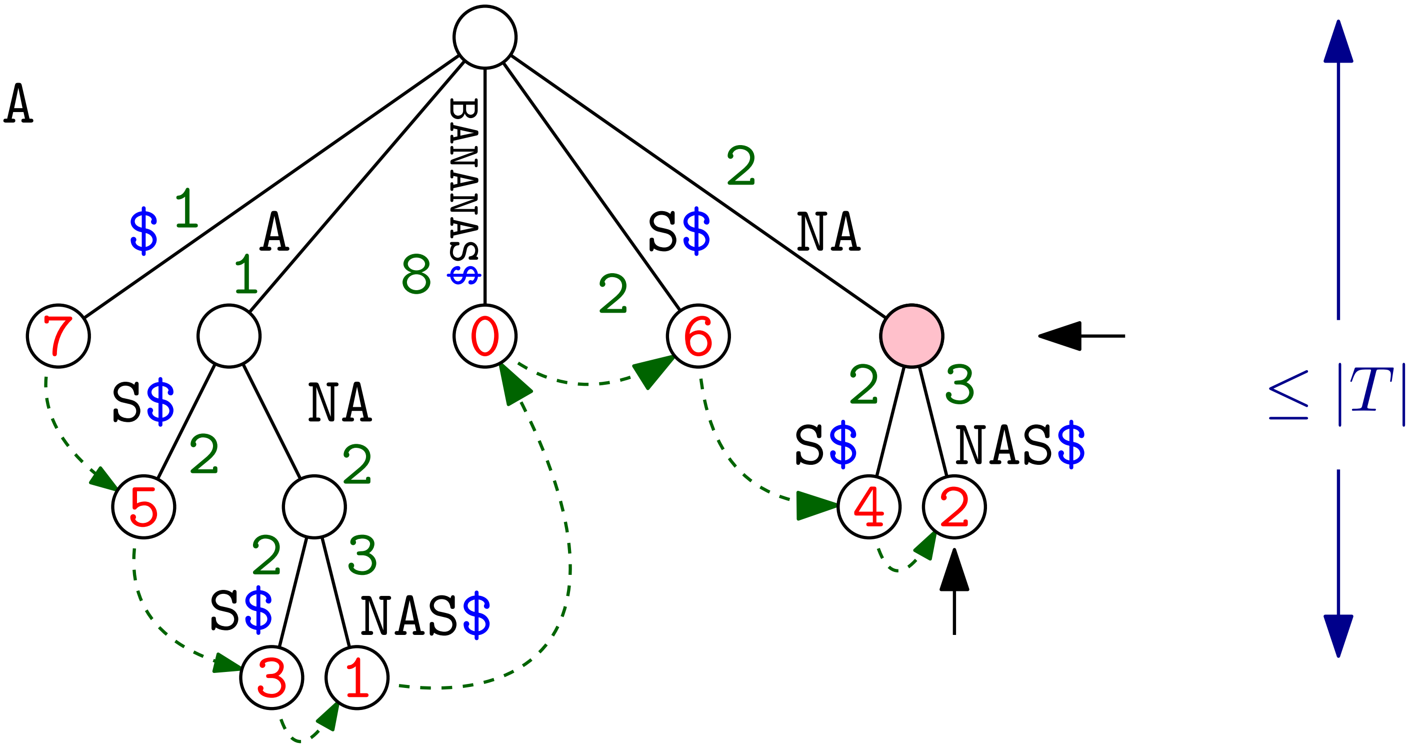
Given an occurrence $T[i:j]$ of P in T, find all other occurrences of P:

- We want to quickly find the node that corresponds to P
- Start from the leaf corresponding to $T[i:]$
- Walk **up** the tree for "$|T| - j$" characters
- This is a **weighted** level ancestor query!

We can answer weighted LA queries in $O(\log \log |T|)$ time!

Applications: Fiding Additional Matches

$P = T[2:3] = \texttt{NA}$



Given an occurrence $T[i:j]$ of P in T, find all other occurrences of P:

- We want to quickly find the node that corresponds to P
- Start from the leaf corresponding to $T[i:]$
- Walk **up** the tree for "$|T| - j$" characters
- This is a **weighted** level ancestor query!
- Link leaves to find the other occurrences in $O(1)$ additional time each

We can answer weighted LA queries in $O(\log\log|T|)$ time!

Applications: Document Retrieval

Preprocess collection of documents $T_1, T_2, \ldots, T_k$ to quickly find all documents that contain a pattern P

Applications: Document Retrieval

Preprocess collection of documents $T_1, T_2, \ldots, T_k$ to quickly find all documents that contain a pattern P

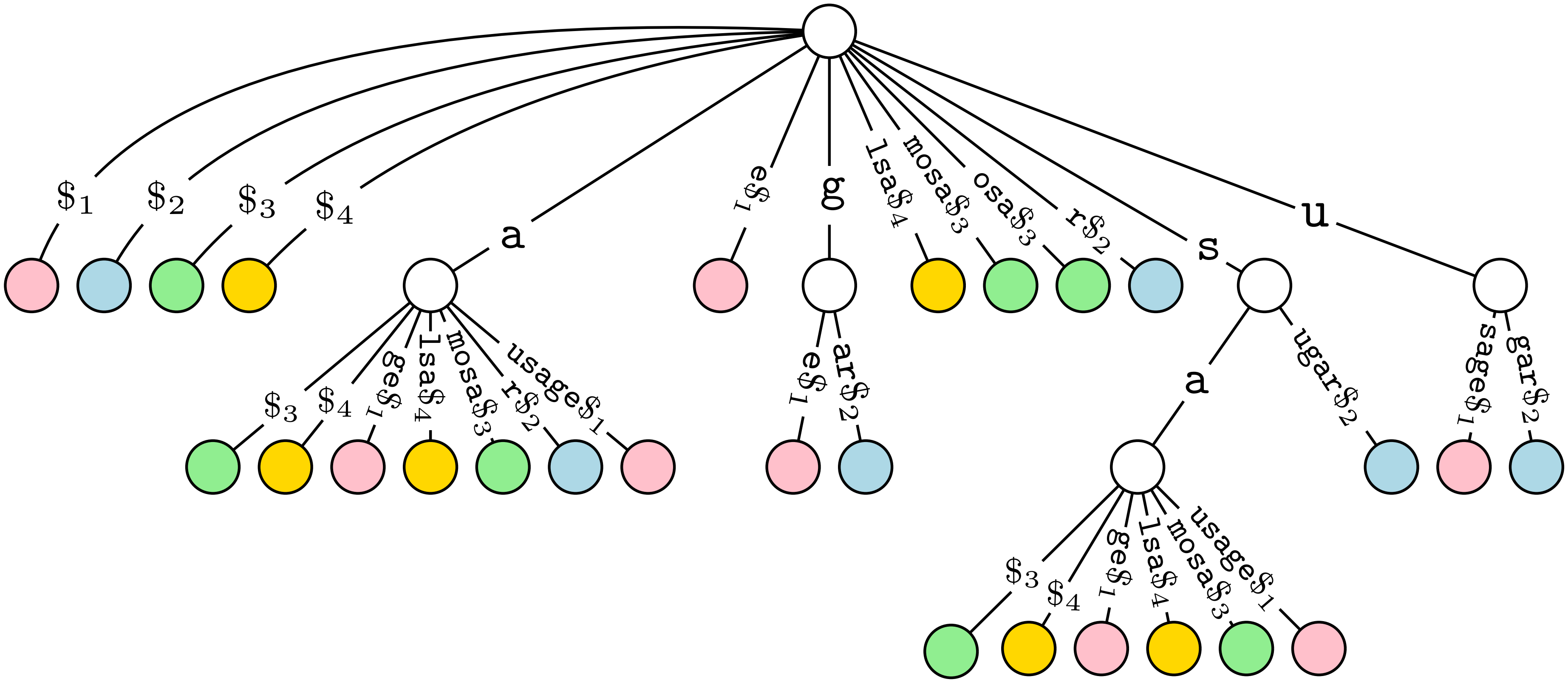
Use the end symbol $\$_i$ for document T_i and build a suffix-tree with the suffxes of all the strings $T_i\$_i$

Applications: Document Retrieval

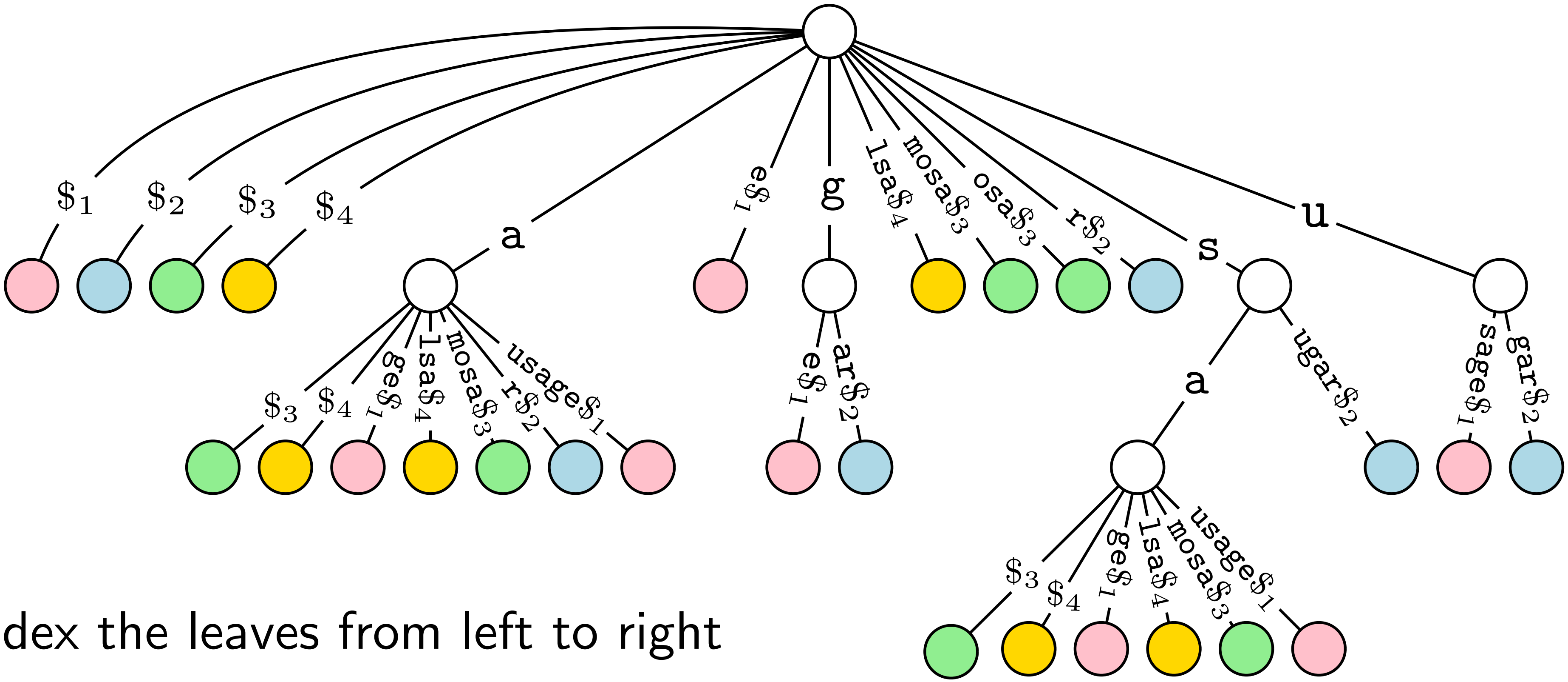
Preprocess collection of documents $T_1, T_2, \ldots, T_k$ to quickly find all documents that contain a pattern P

Use the end symbol $\$_i$ for document T_i and build a suffix-tree with the suffxes of all the strings $T_i\$_i$

sausage$\$_1$ sugar$\$_2$ samosa$\$_3$ salsa$\$_4$

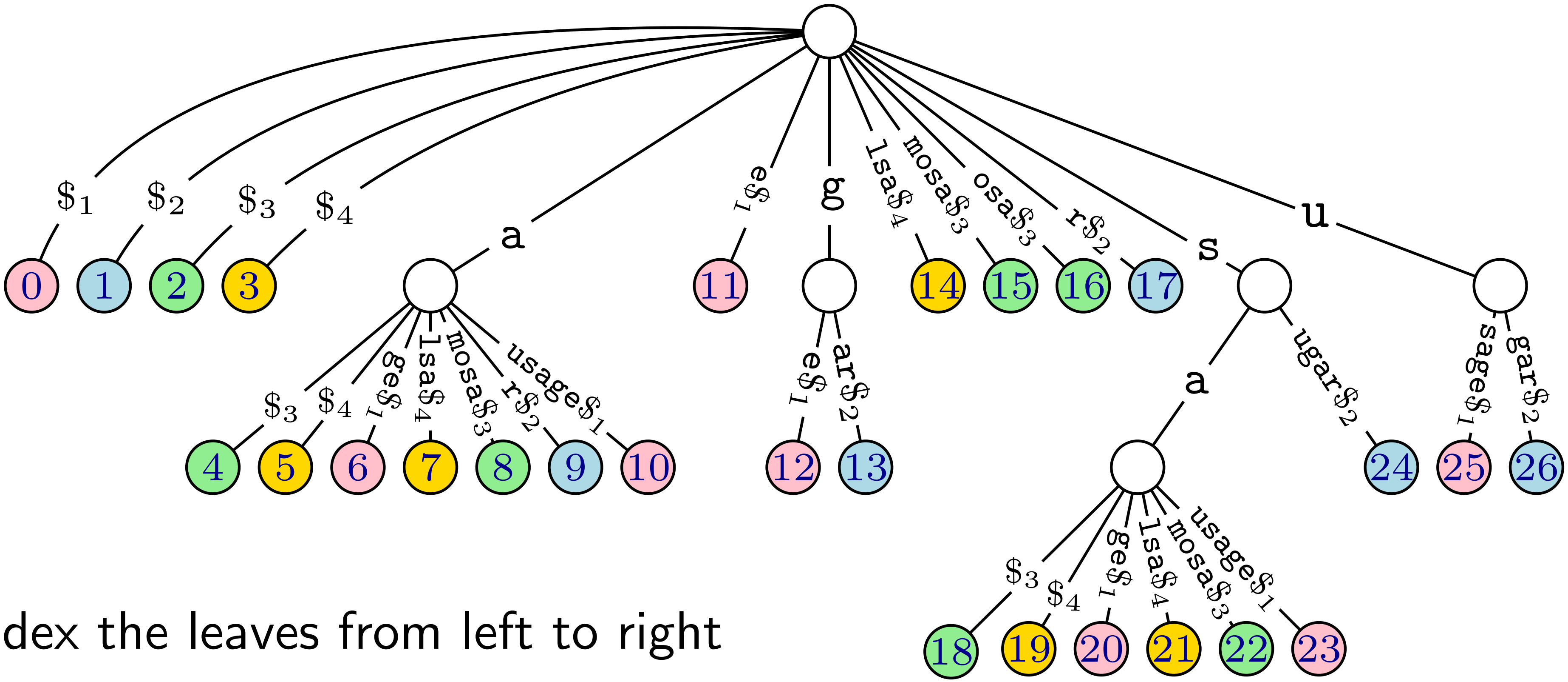


Applications: Document Retrieval



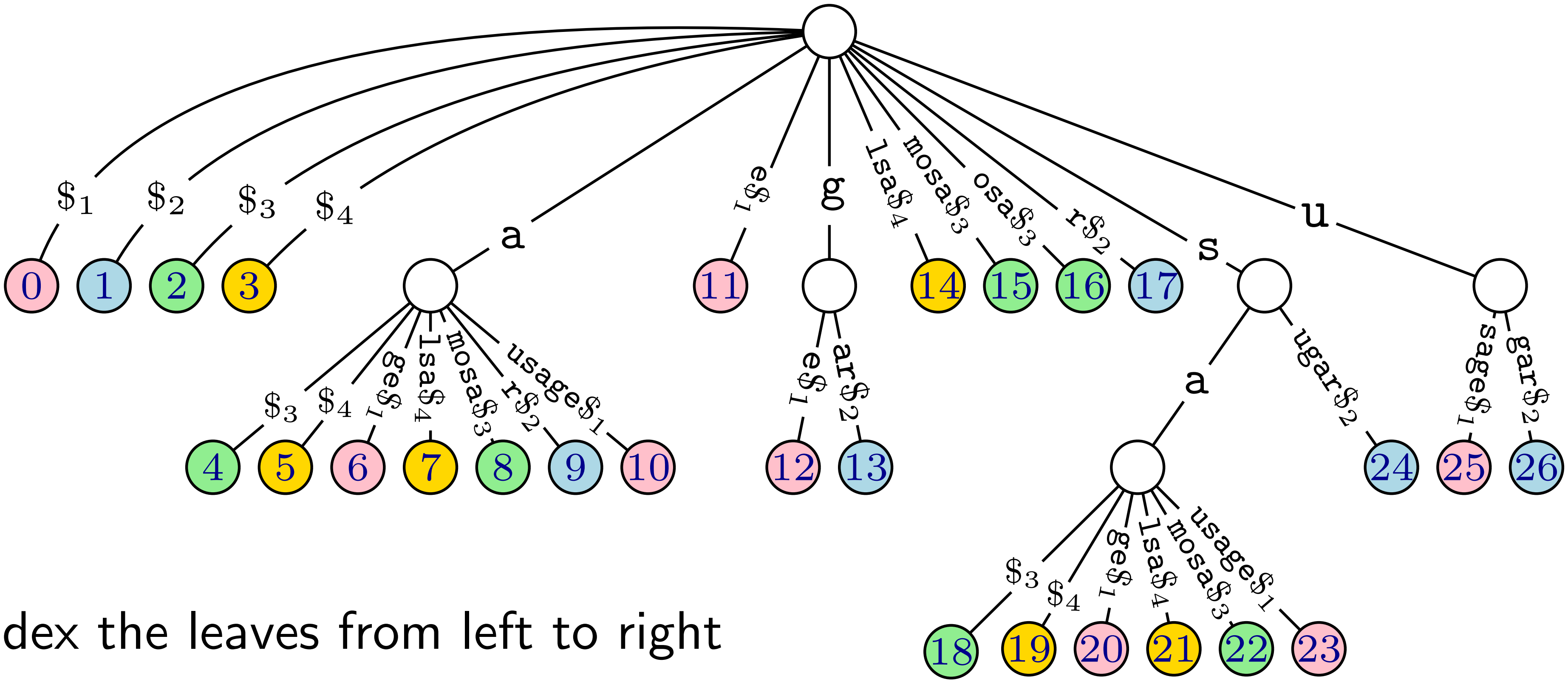
Index the leaves from left to right

Applications: Document Retrieval



Index the leaves from left to right

Applications: Document Retrieval

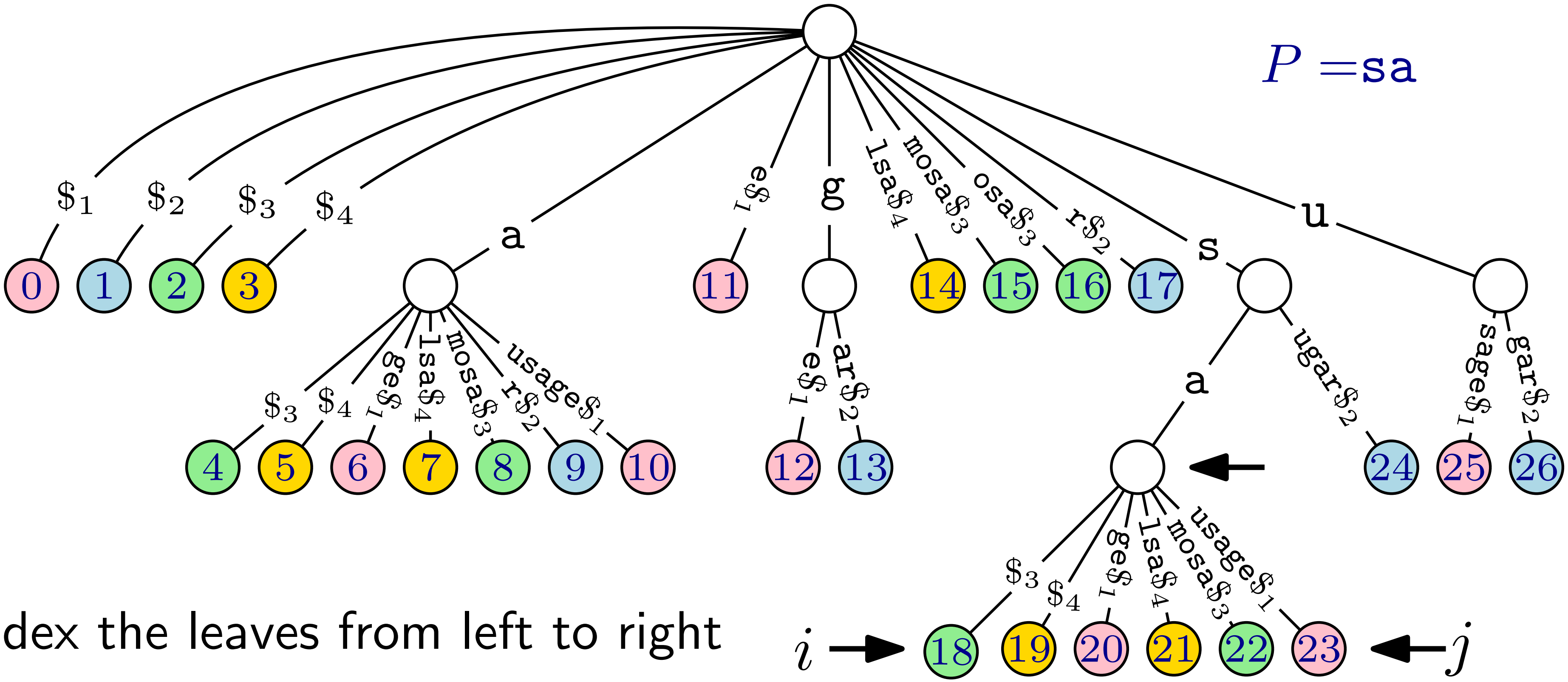


Index the leaves from left to right

Store an array A where $A[i]$ poinst to the document of leaf i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
🟥	🟦	🟩	🟨	🟩	🟨	🟥	🟨	🟩	🟦	🟥	🟥	🟥	🟦	🟨	🟩	🟩	🟦	🟩	🟨	🟥	🟨	🟩	🟥	🟦	🟥	🟦

Applications: Document Retrieval



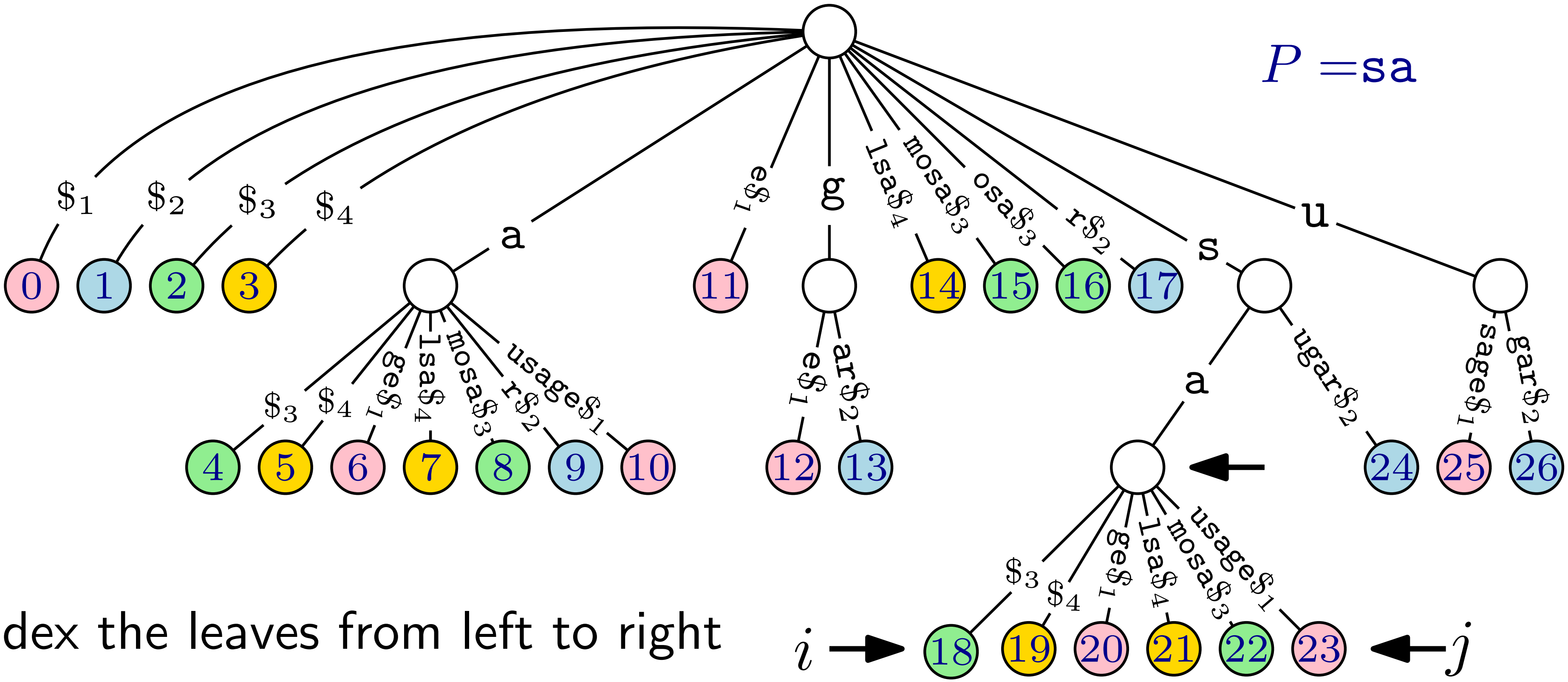
Index the leaves from left to right

Store an array A where $A[i]$ poinst to the document of leaf i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
🟥	🟦	🟩	🟨	🟩	🟨	🟥	🟨	🟩	🟦	🟥	🟥	🟥	🟦	🟨	🟩	🟩	🟦	🟩	🟨	🟥	🟨	🟩	🟥	🟦	🟥	🟦

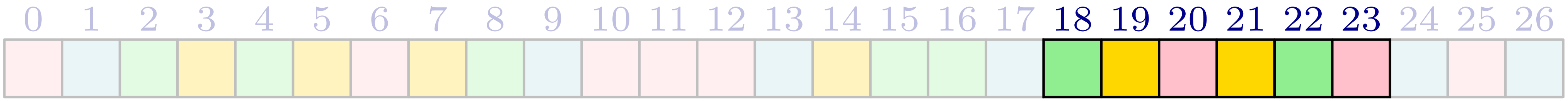
Searching for a pattern P returns the interval $A[i:j]$ containing all and only the leaves corresponding to the matches of P

Applications: Document Retrieval



Index the leaves from left to right

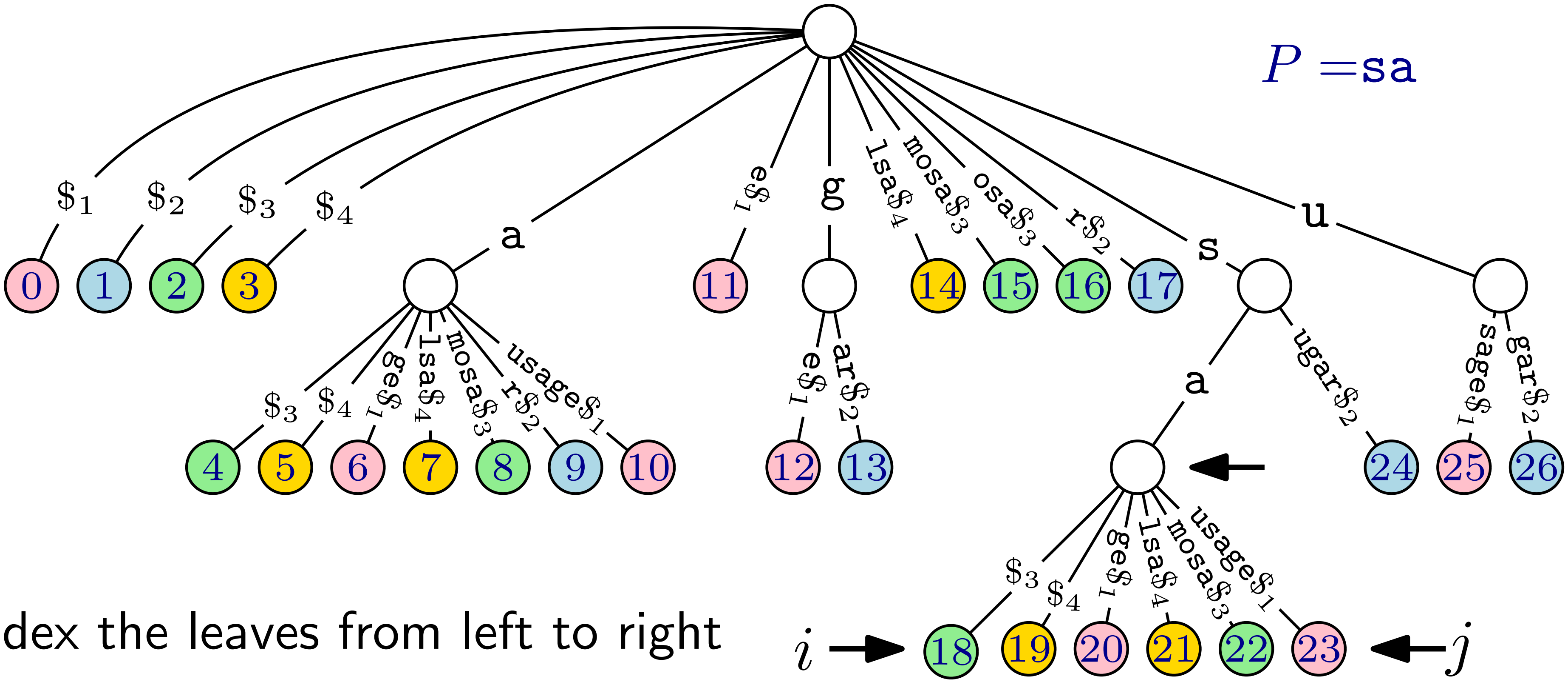
Store an array A where $A[i]$ poinst to the document of leaf i



Searching for a pattern P returns the interval $A[i:j]$ containing all and only the leaves corresponding to the matches of P

Find all distinct documents (colors) in $A[i:j]$

Applications: Document Retrieval



Index the leaves from left to right

Store an array A where $A[i]$ poinst to the document of leaf i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
🟥	🟦	🟩	🟨	🟩	🟨	🟥	🟨	🟩	🟦	🟥	🟥	🟥	🟦	🟨	🟩	🟩	🟦	🟩	🟨	🟥	🟨	🟩	🟥	🟦	🟥	🟦

Searching for a pattern P retu

only the leaves corresponding t

Find all distinct documents (c

Time:

$O(|P| + \log|\Sigma| + \#$ retrieved documents$)$

via range minimum queries

Constructing Suffix Trees & Suffix Arrays

Suffix Arrays & Suffix Trees

$T =$ BANANAS

Sort all suffixes along with their start index

0	BANANAS$
1	ANANAS$
2	NANAS$
3	ANAS$
4	NAS$
5	AS$
6	S$
7	$

Suffix Arrays & Suffix Trees

$T = \texttt{BANANAS}$

Sort all suffixes along with their start index

7	\$
1	ANANAS\$
3	ANAS\$
5	AS\$
0	BANANAS\$
2	NANAS\$
4	NAS\$
6	S\$

Suffix Arrays & Suffix Trees

$T =$ BANANAS

Sort all suffixes along with their start index

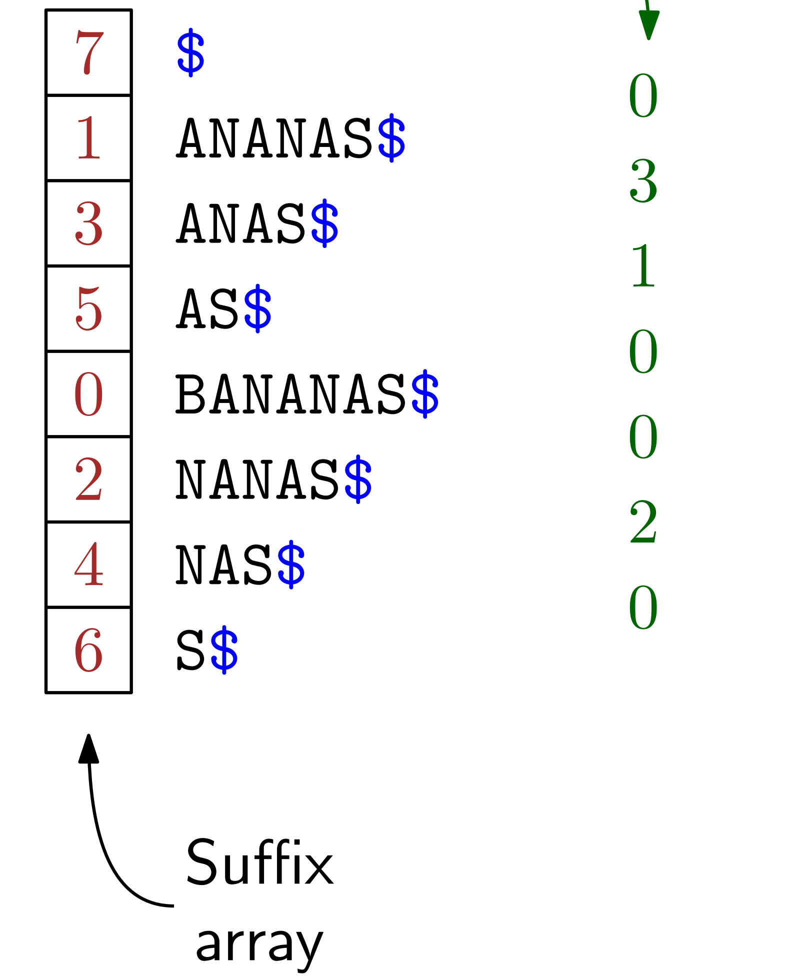
7	$
1	ANANAS$
3	ANAS$
5	AS$
0	BANANAS$
2	NANAS$
4	NAS$
6	S$

Suffix array

Suffix Arrays & Suffix Trees

$T =$ `BANANAS`

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



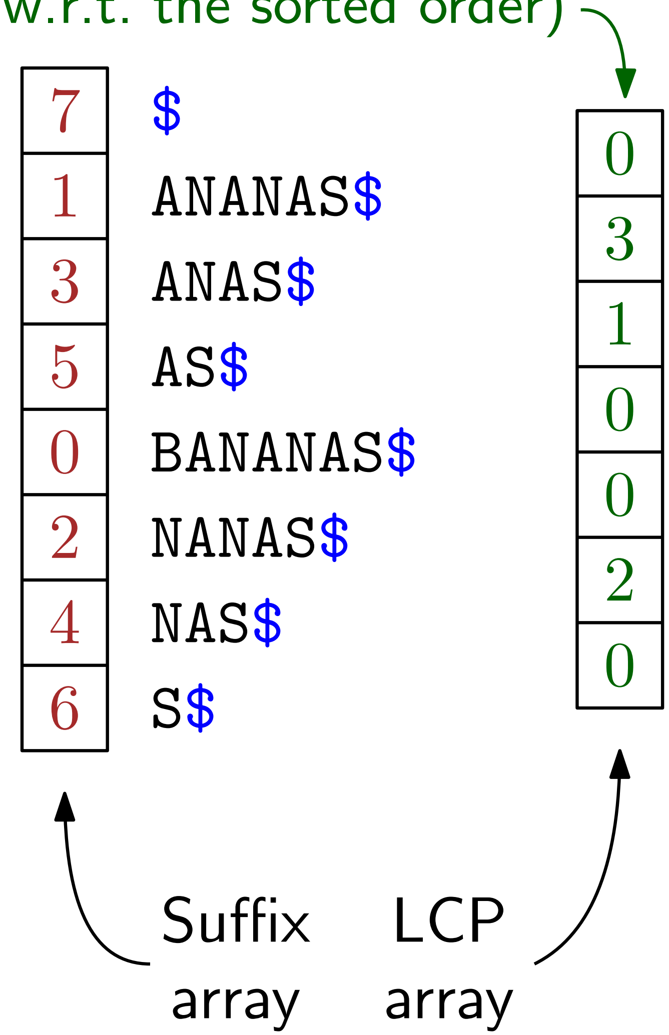
7	$	
		0
1	ANANAS$	
		3
3	ANAS$	
		1
5	AS$	
		0
0	BANANAS$	
		0
2	NANAS$	
		2
4	NAS$	
		0
6	S$	

Suffix array

Suffix Arrays & Suffix Trees

$T =$ `BANANAS`

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



7	$
1	ANANAS$
3	ANAS$
5	AS$
0	BANANAS$
2	NANAS$
4	NAS$
6	S$

0
3
1
0
0
2
0

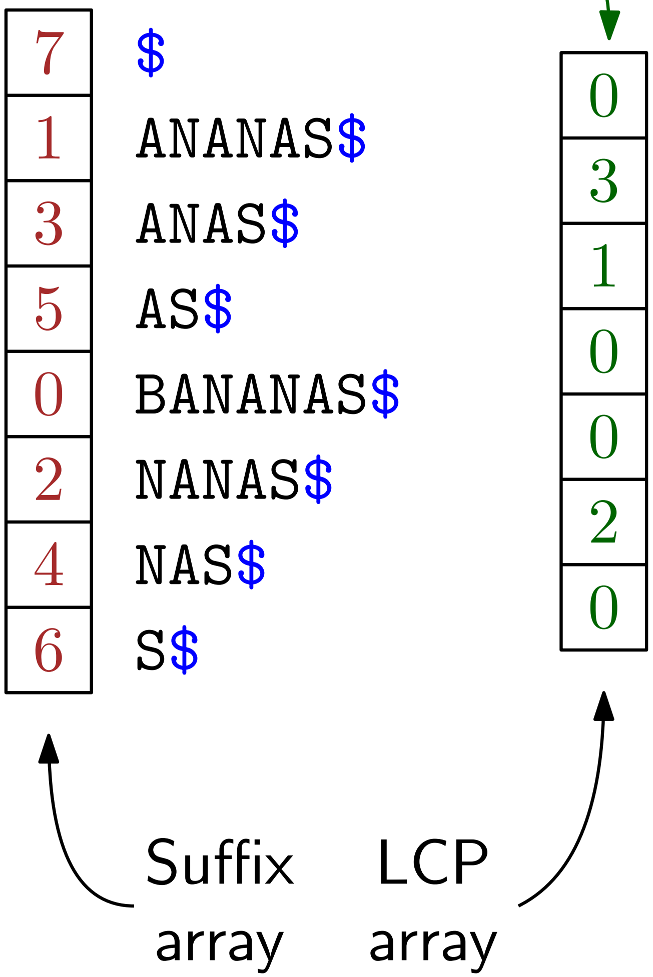
Suffix array

LCP array

Suffix Arrays & Suffix Trees

$T =$ BANANAS

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



7	$	0
1	ANANAS$	3
3	ANAS$	1
5	AS$	0
0	BANANAS$	0
2	NANAS$	2
4	NAS$	0
6	S$	

Suffix array

LCP array

We can construct a suffix tree from the Suffix and LCP arrays

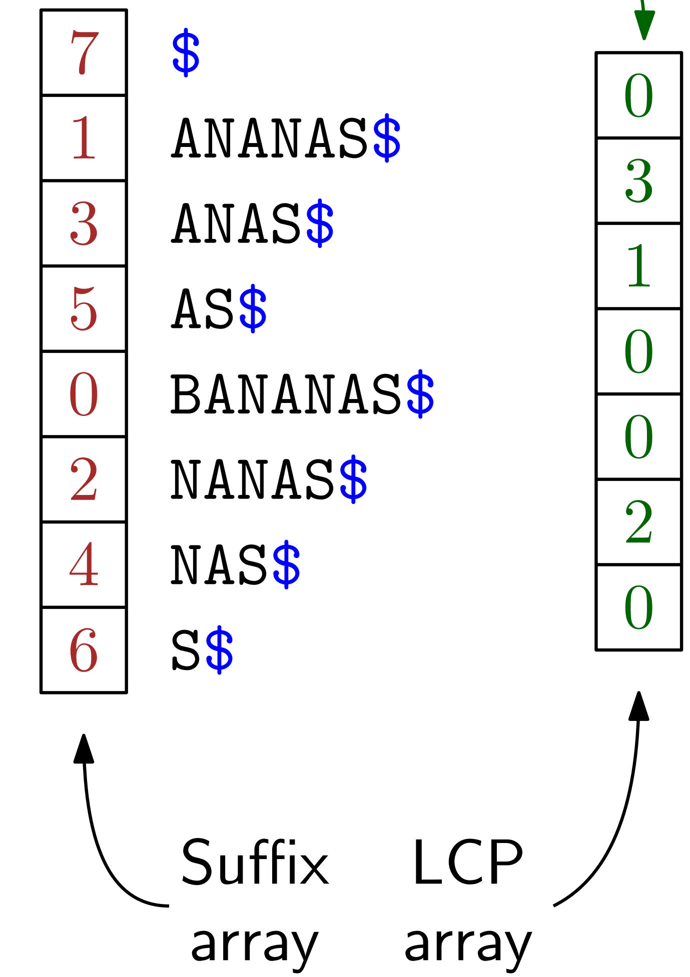
A construction similar to the one of cartesian trees yields the subtree of branching vertices

0	3	1	0	0	2	0

Suffix Arrays & Suffix Trees

$T =$ BANANAS

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



Suffix array	
7	$
1	ANANAS$
3	ANAS$
5	AS$
0	BANANAS$
2	NANAS$
4	NAS$
6	S$

LCP array
0
3
1
0
0
2
0

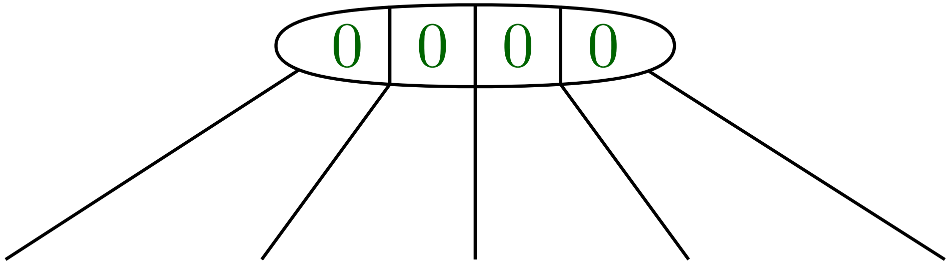
Suffix array

LCP array

We can construct a suffix tree from the Suffix and LCP arrays

A construction similar to the one of cartesian trees yields the subtree of branching vertices

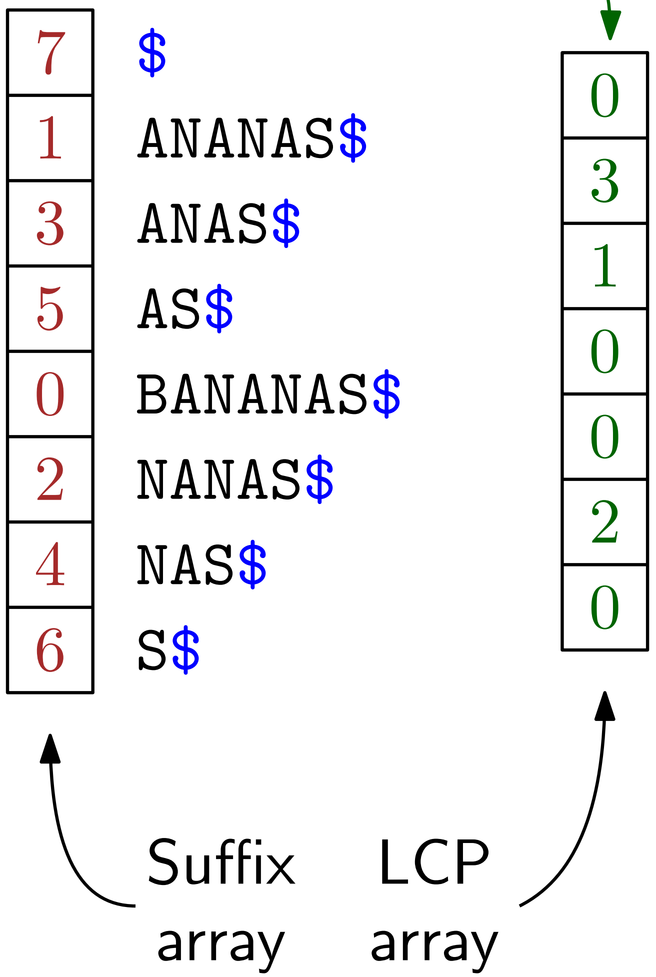
0	3	1	0	0	2	0



Suffix Arrays & Suffix Trees

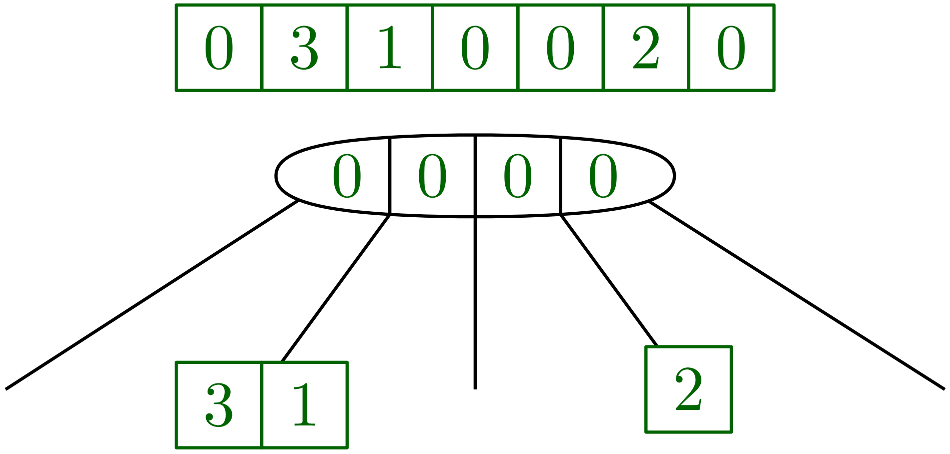
$T =$ `BANANAS`

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



We can construct a suffix tree from the Suffix and LCP arrays

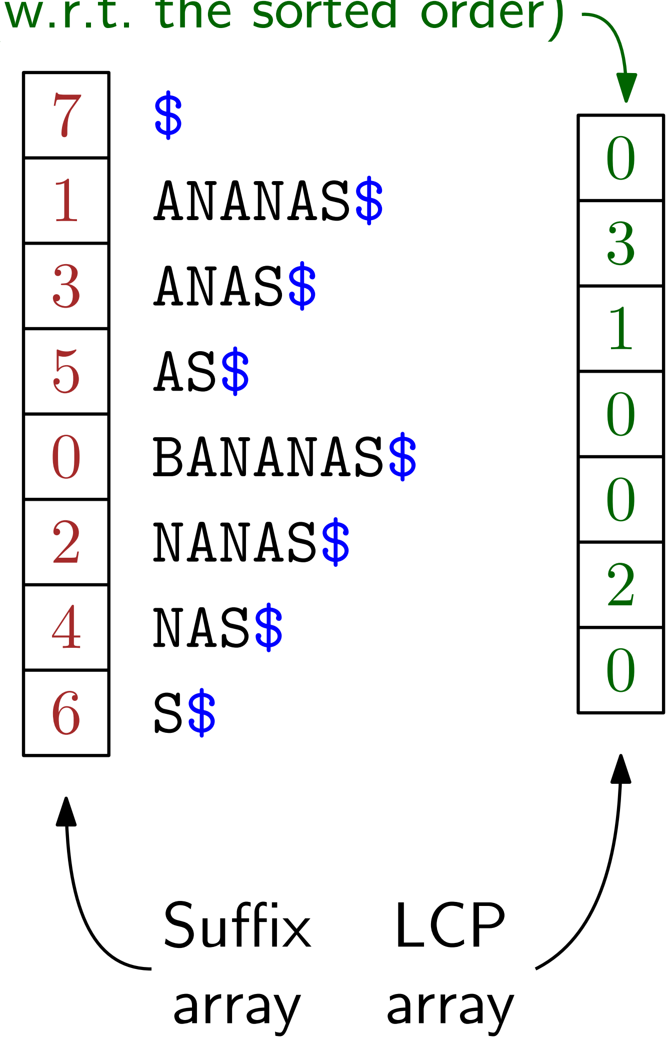
A construction similar to the one of cartesian trees yields the subtree of branching vertices



Suffix Arrays & Suffix Trees

$T =$ BANANAS

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



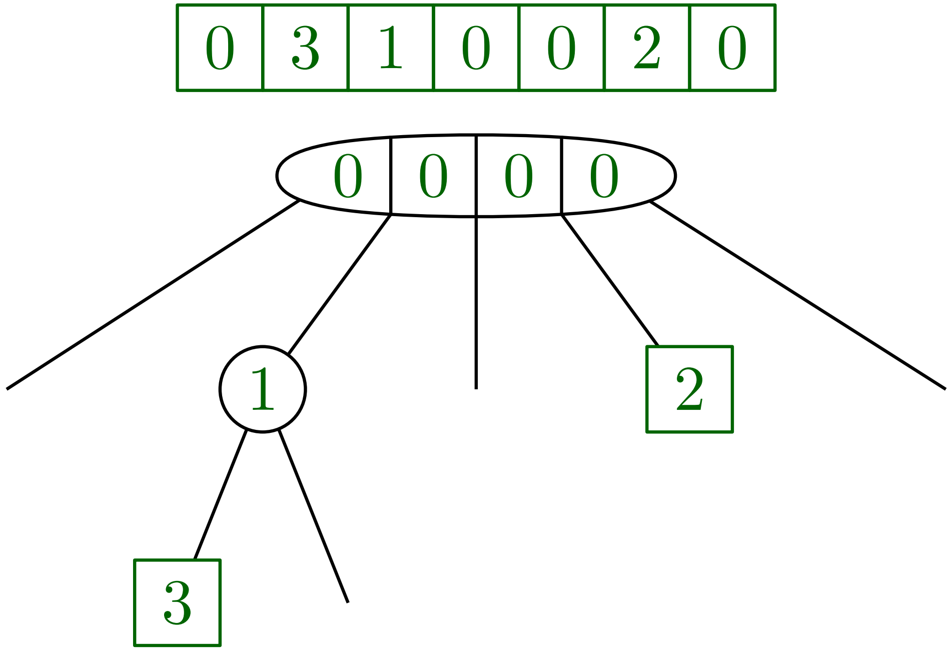
Suffix array		LCP array
7	$	0
1	ANANAS$	3
3	ANAS$	1
5	AS$	0
0	BANANAS$	0
2	NANAS$	2
4	NAS$	0
6	S$	

Suffix array

LCP array

We can construct a suffix tree from the Suffix and LCP arrays

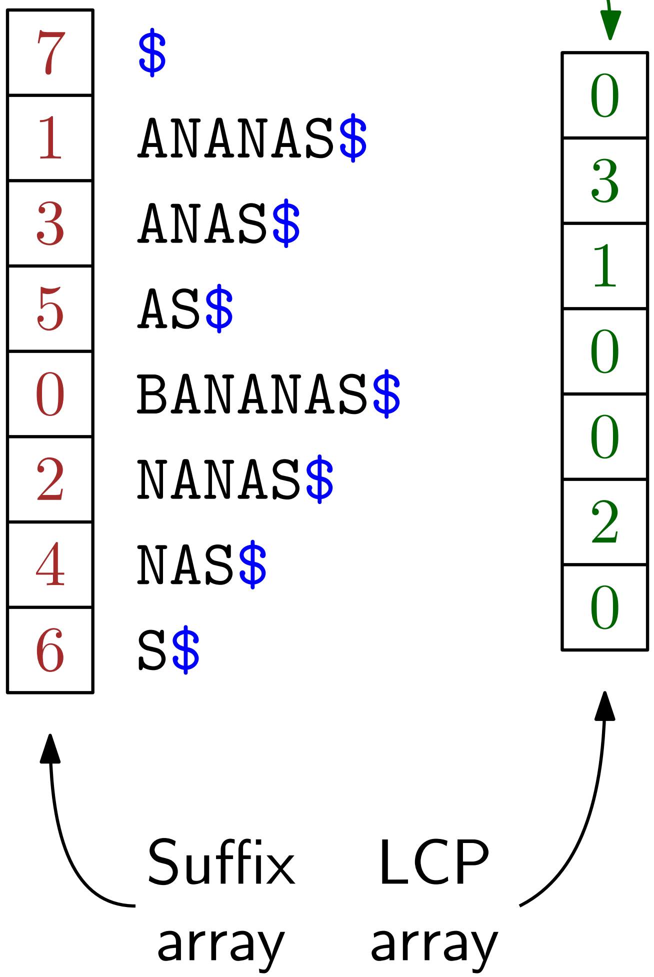
A construction similar to the one of cartesian trees yields the subtree of branching vertices



Suffix Arrays & Suffix Trees

$T = \texttt{BANANAS}$

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



7	$
1	ANANAS$
3	ANAS$
5	AS$
0	BANANAS$
2	NANAS$
4	NAS$
6	S$

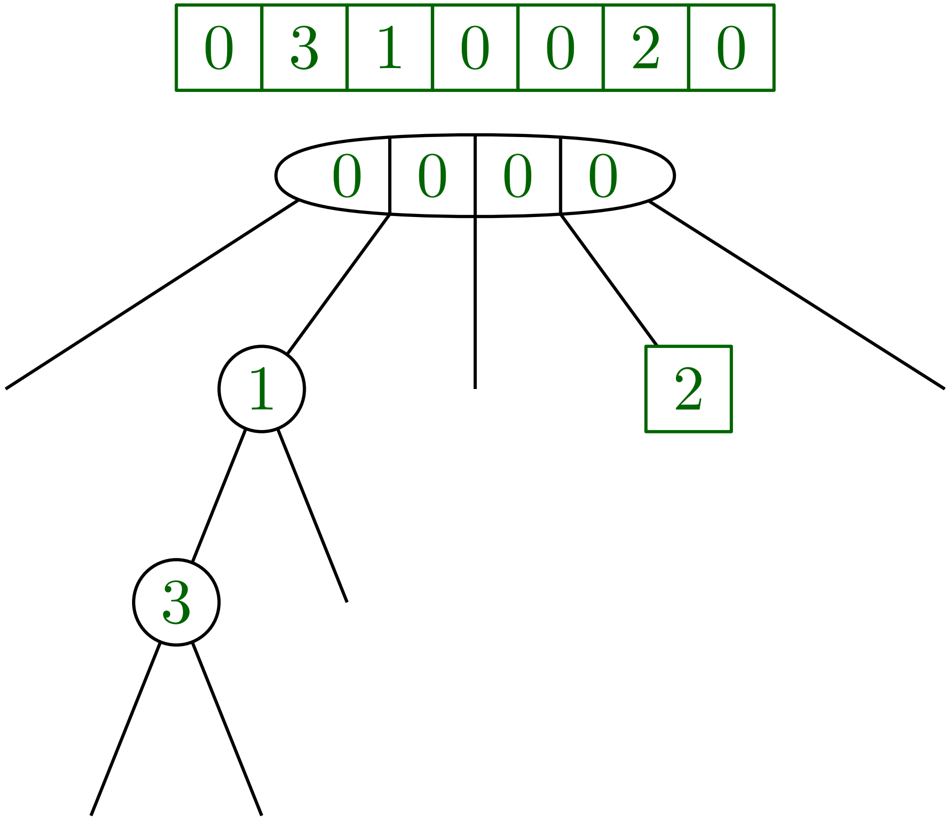
0
3
1
0
0
2
0

Suffix array

LCP array

We can construct a suffix tree from the Suffix and LCP arrays

A construction similar to the one of cartesian trees yields the subtree of branching vertices

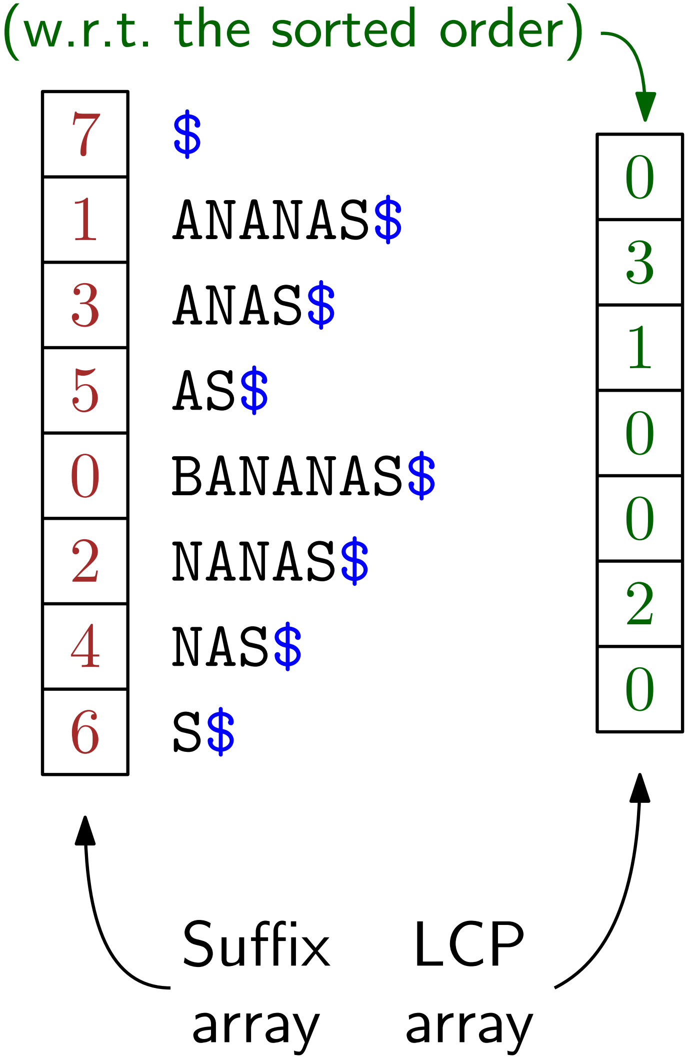


0	3	1	0	0	2	0

Suffix Arrays & Suffix Trees

$T =$ BANANAS

Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



7	$
1	ANANAS$
3	ANAS$
5	AS$
0	BANANAS$
2	NANAS$
4	NAS$
6	S$

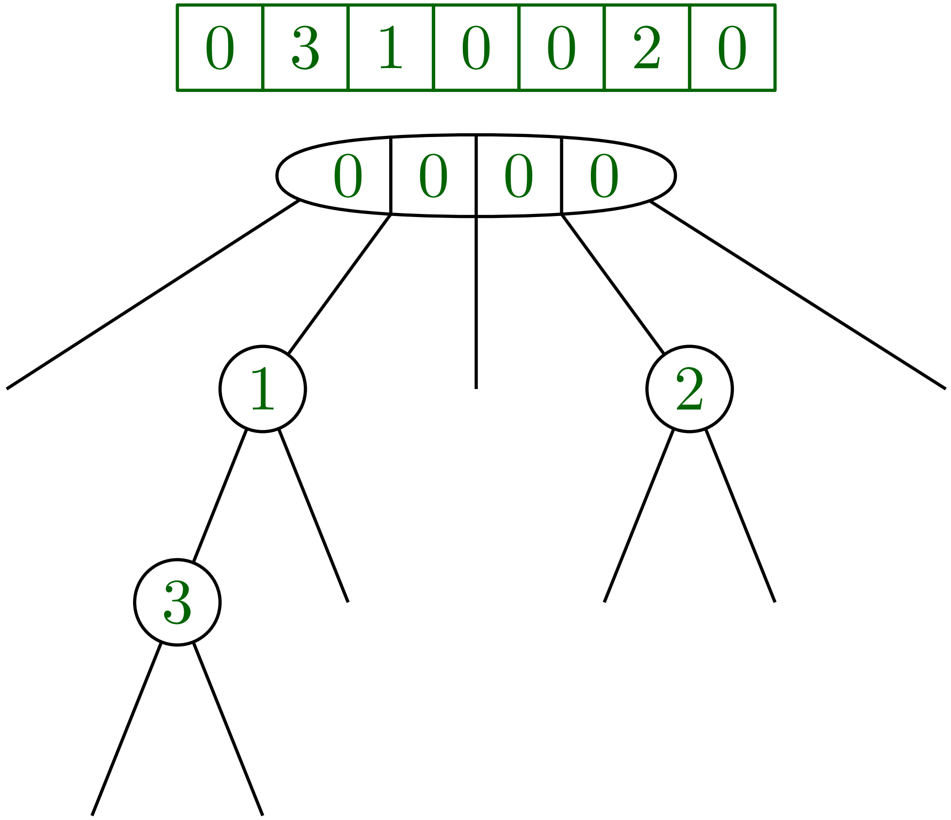
0
3
1
0
0
2
0

Suffix array

LCP array

We can construct a suffix tree from the Suffix and LCP arrays

A construction similar to the one of cartesian trees yields the subtree of branching vertices

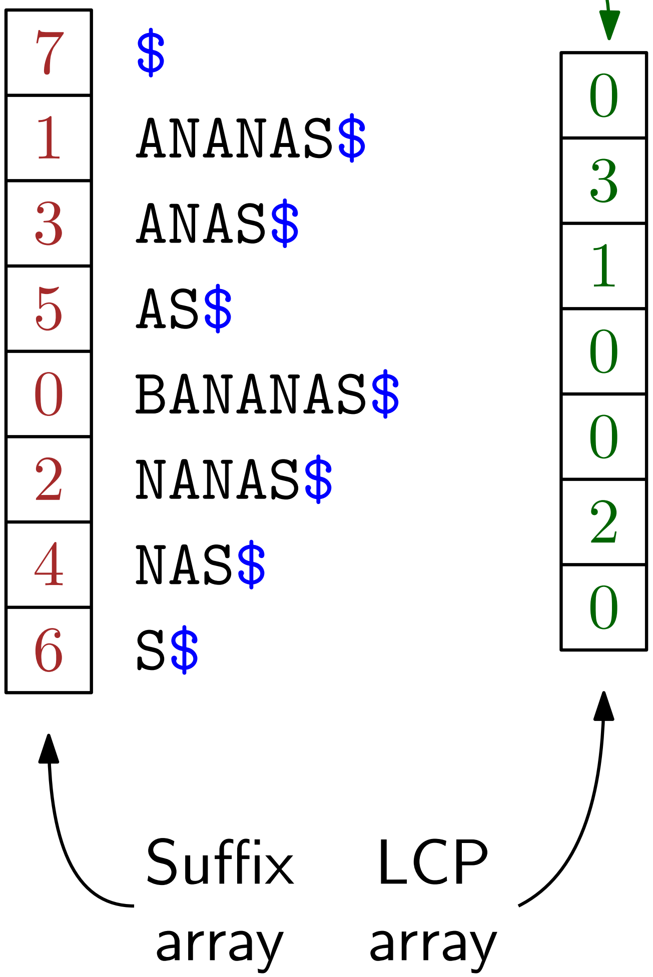


0	3	1	0	0	2	0

Suffix Arrays & Suffix Trees

$T =$ `BANANAS`

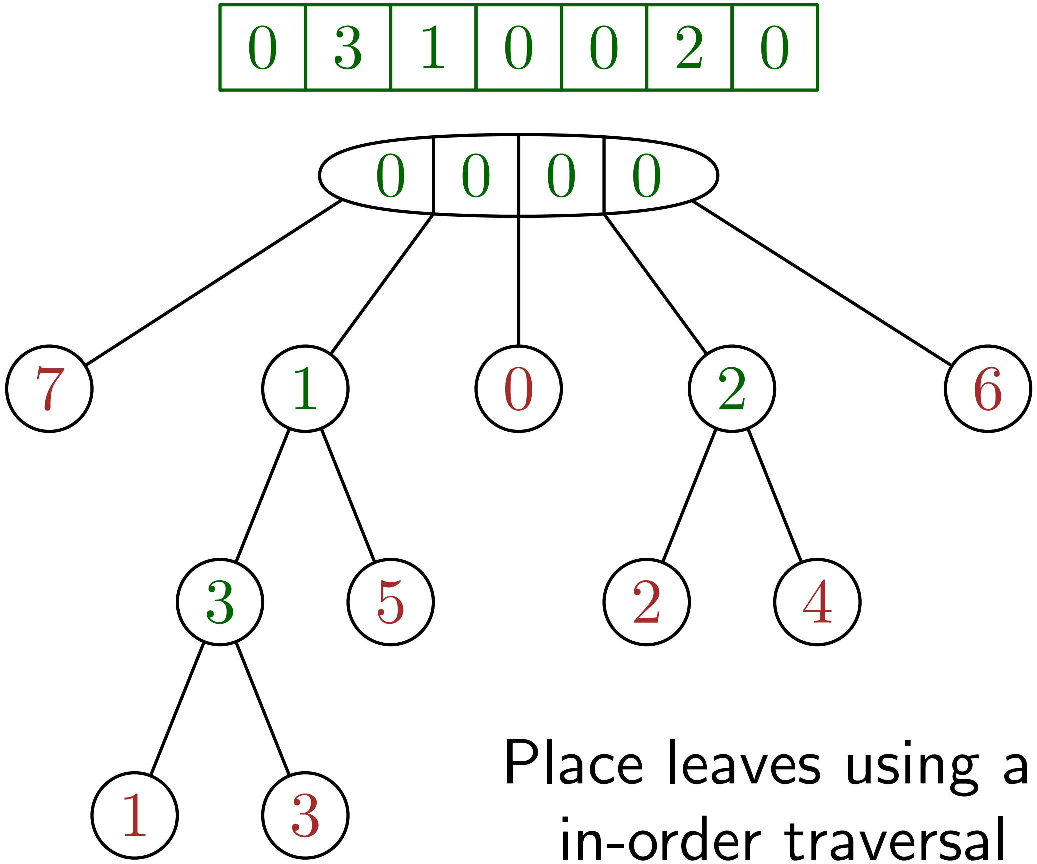
Length of the longest common prefix between adjacent suffixes (w.r.t. the sorted order)



Suffix array		LCP array
7	$	0
1	ANANAS$	3
3	ANAS$	1
5	AS$	0
0	BANANAS$	0
2	NANAS$	2
4	NAS$	0
6	S$	

We can construct a suffix tree from the Suffix and LCP arrays

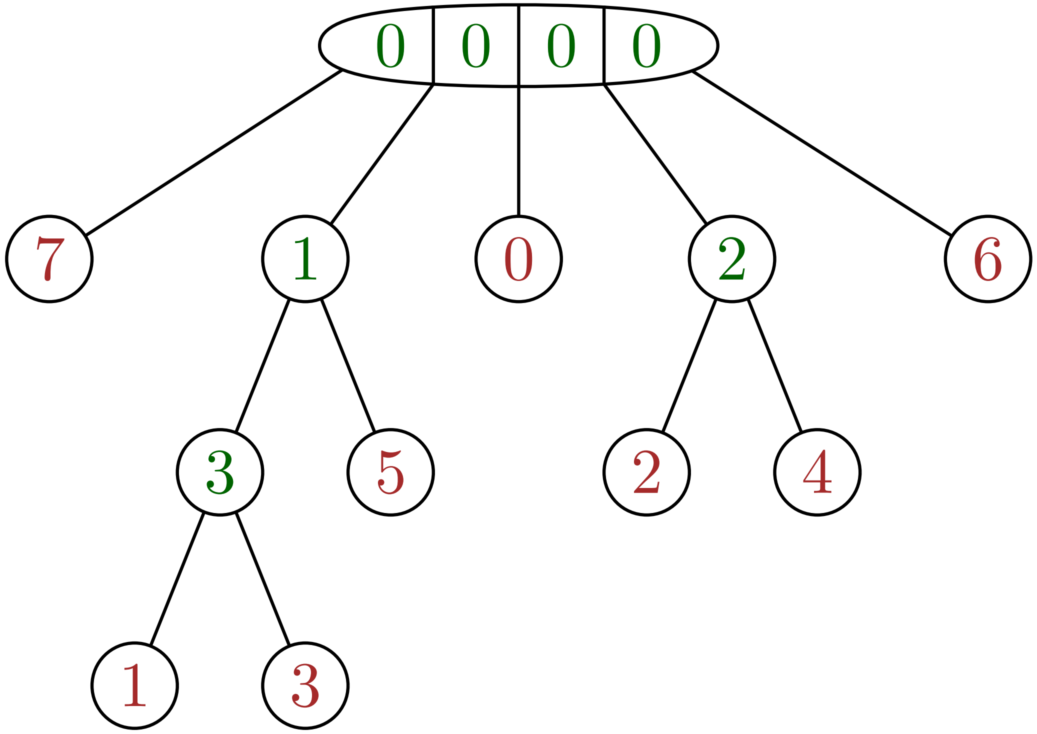
A construction similar to the one of cartesian trees yields the subtree of branching vertices



Place leaves using an in-order traversal

Suffix Arrays & Suffix Trees

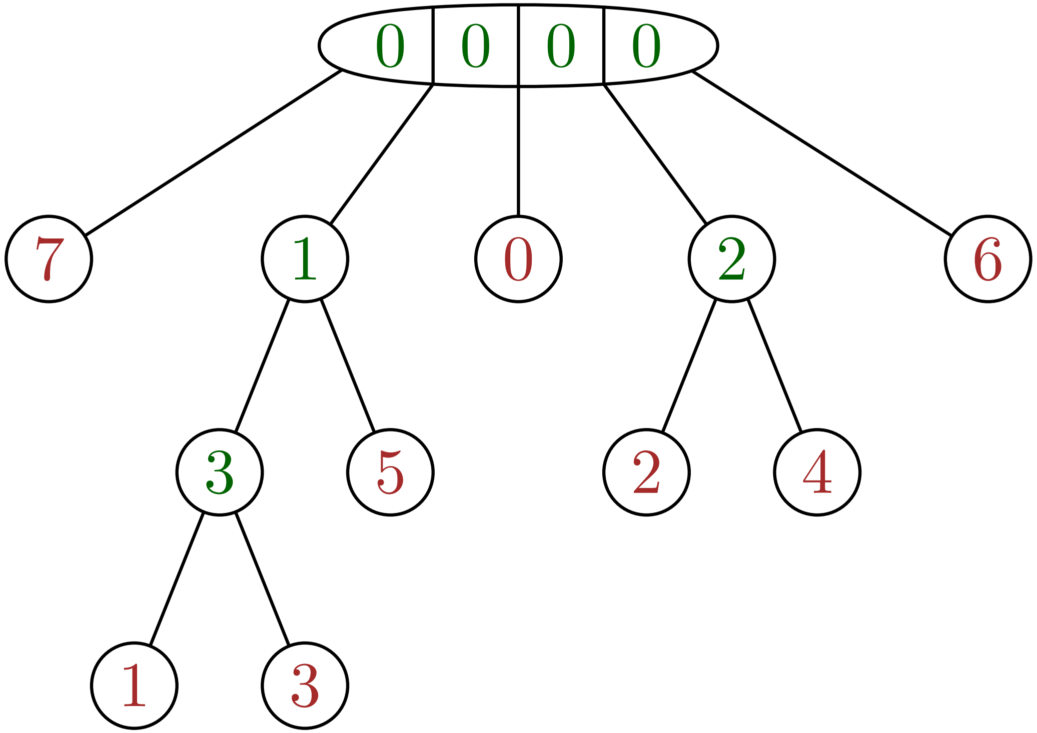
Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

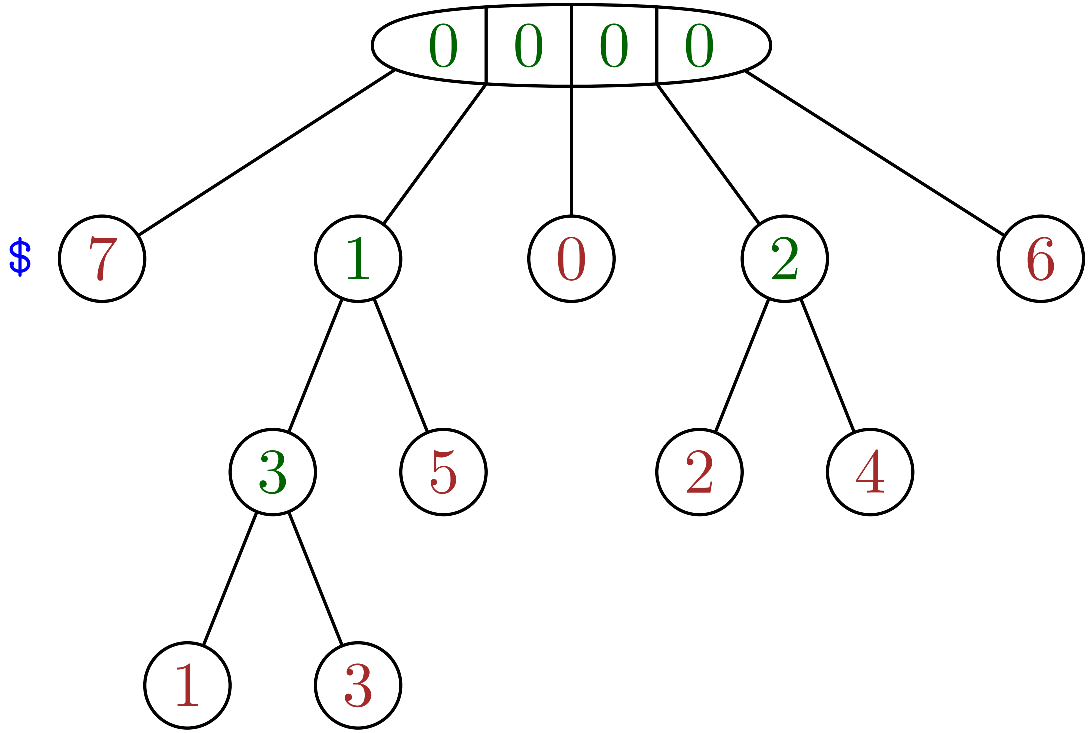
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

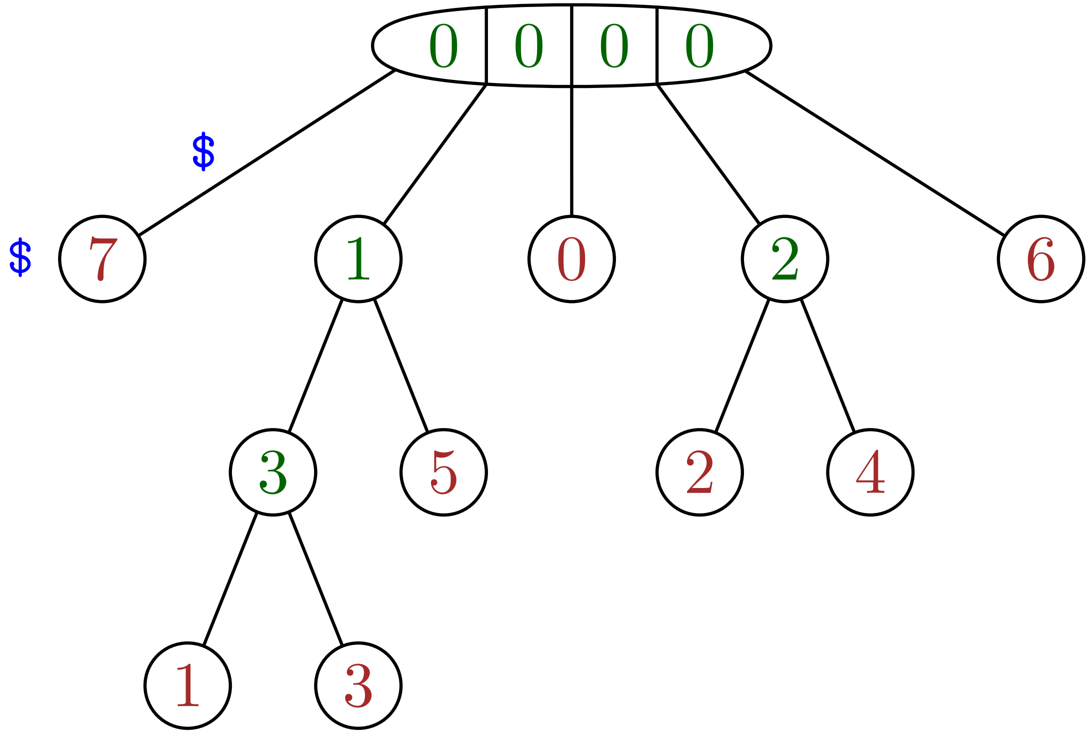
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

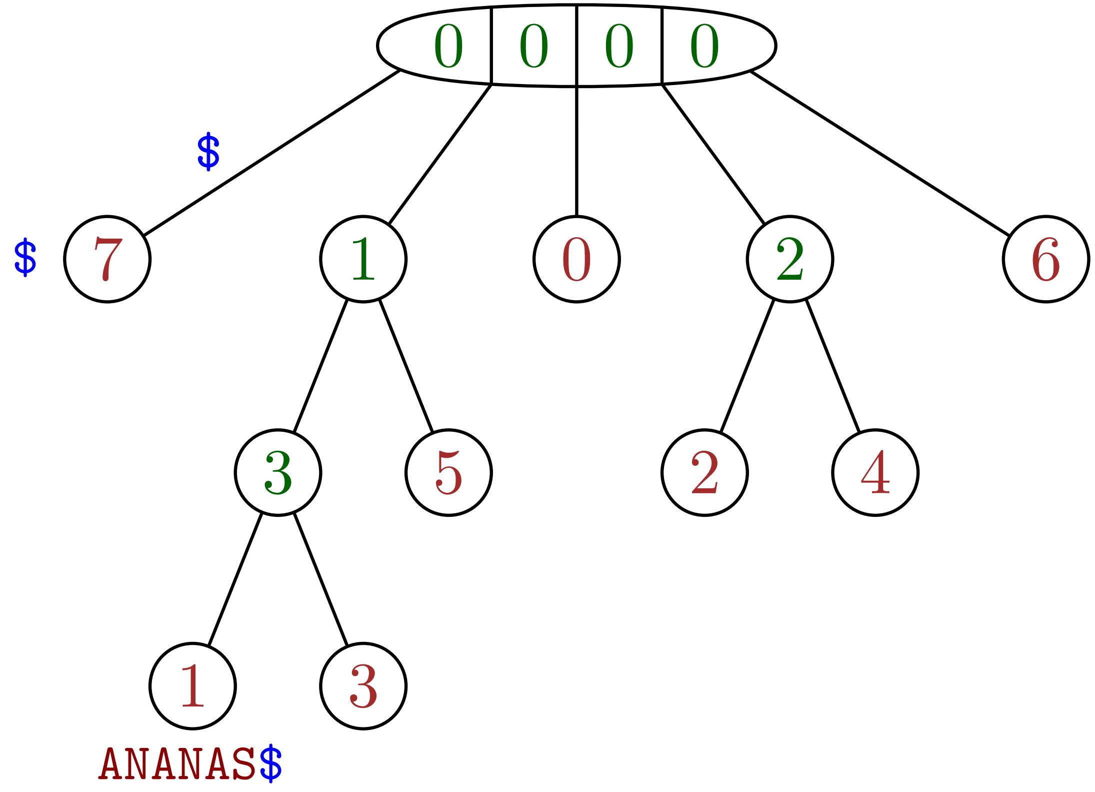
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

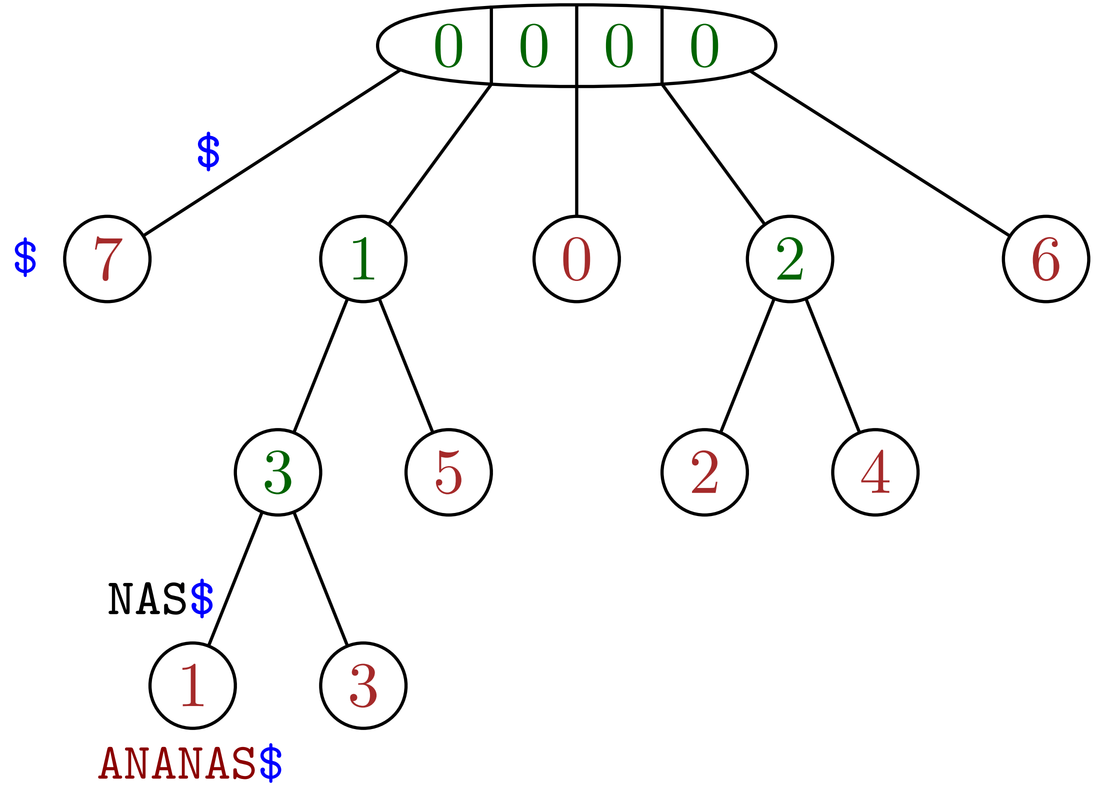
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

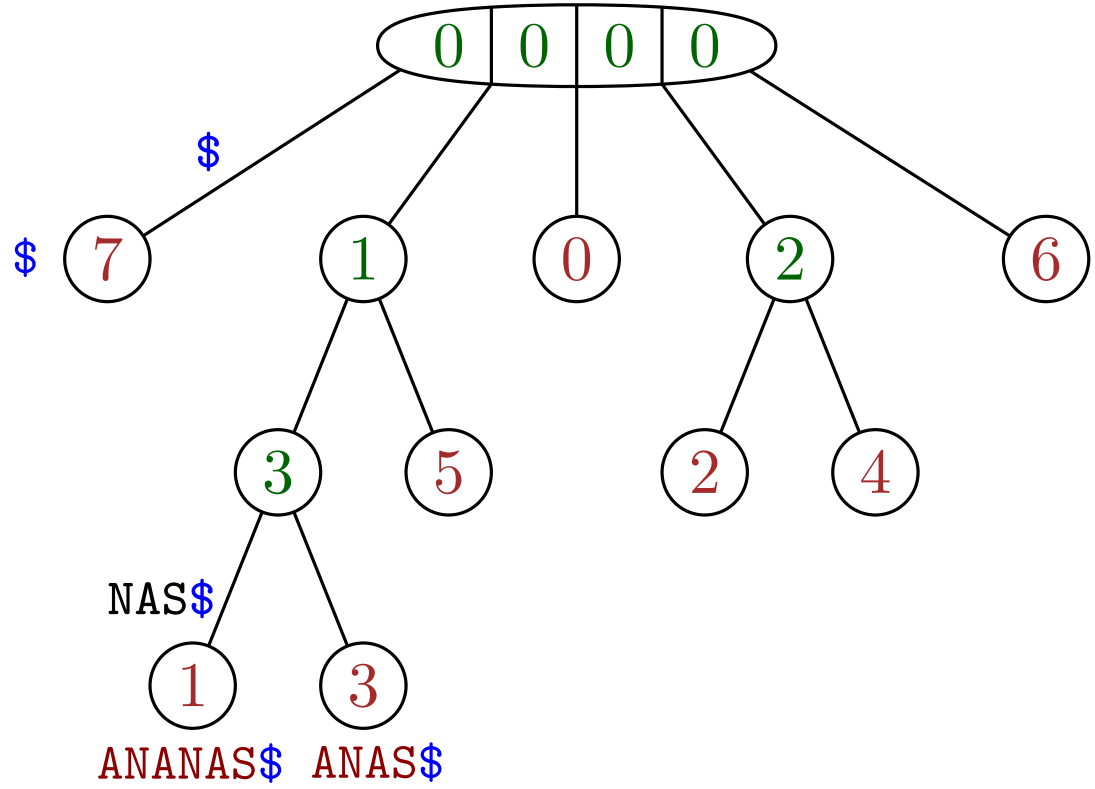
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

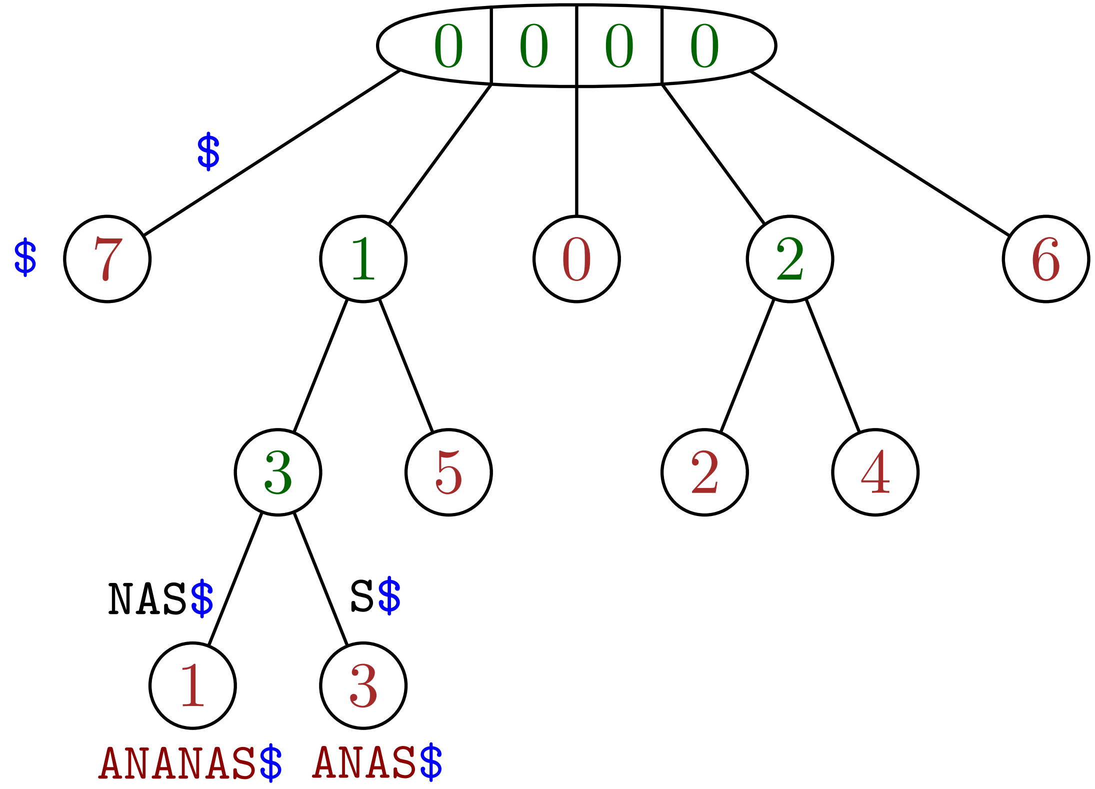
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

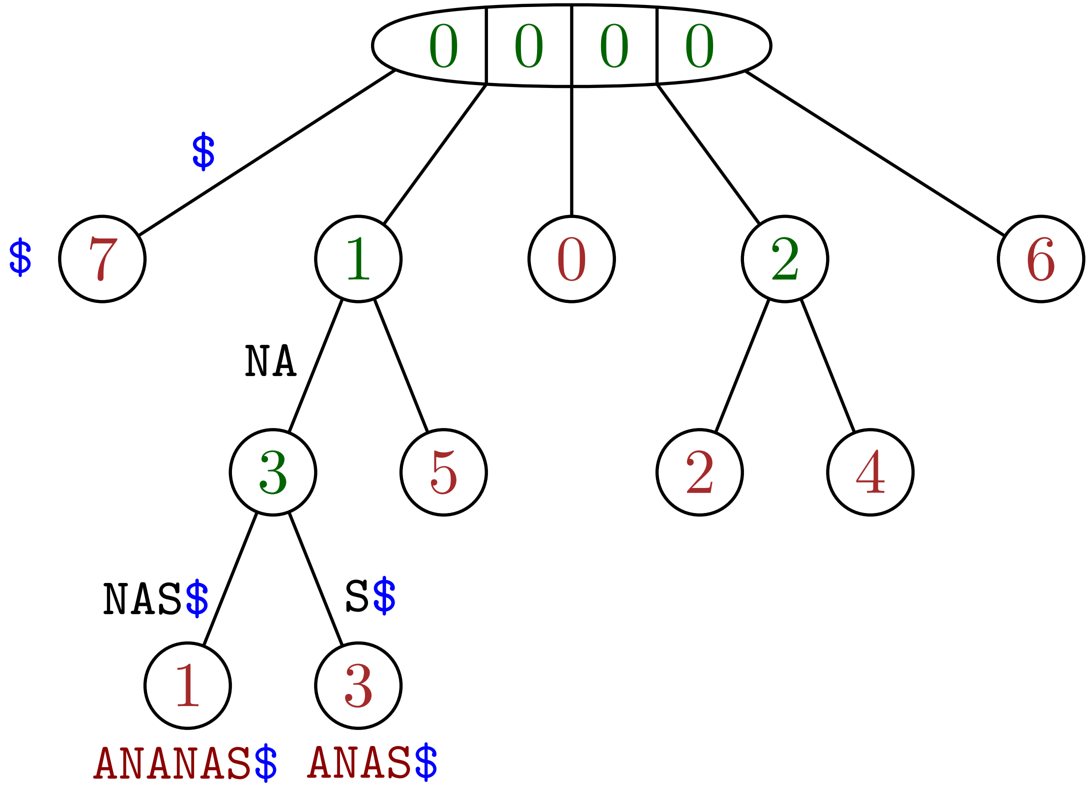
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

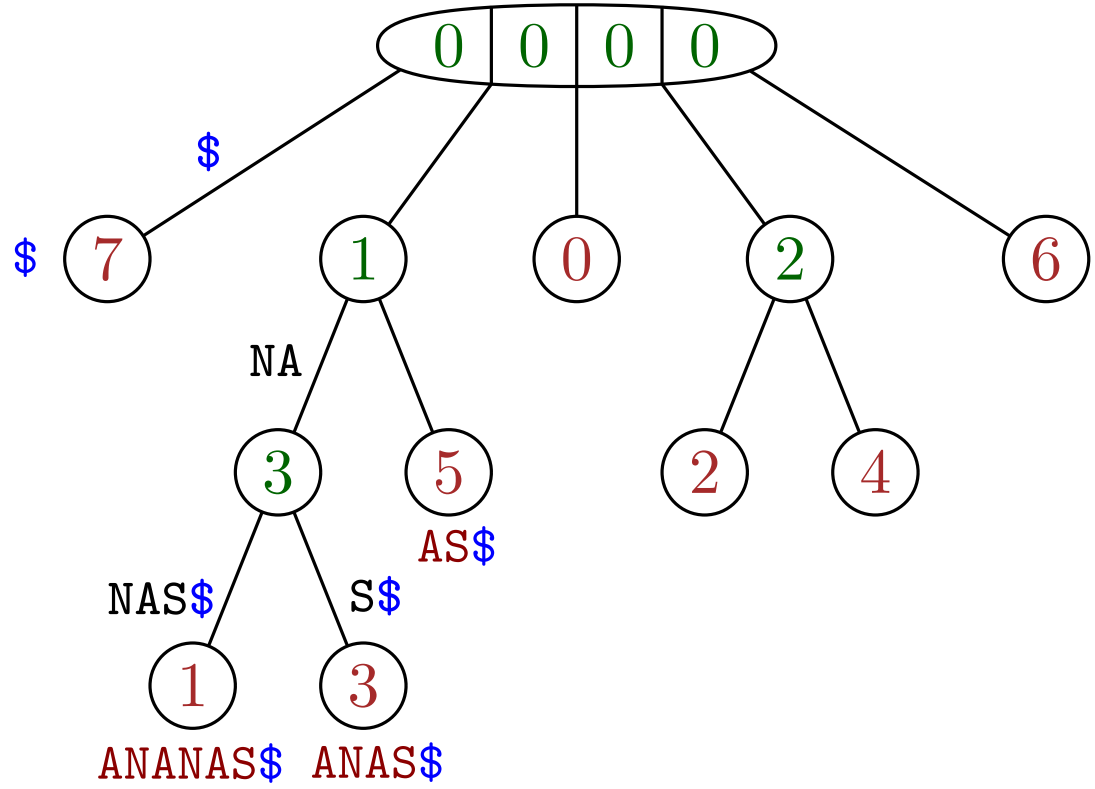
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

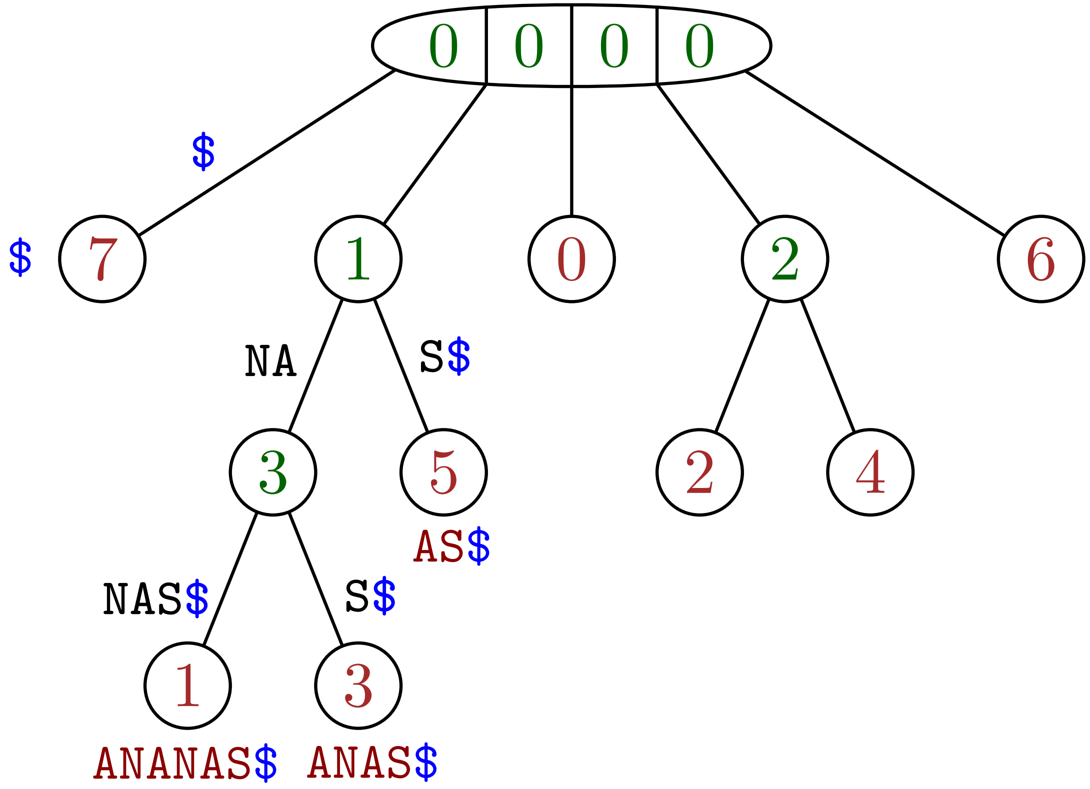
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

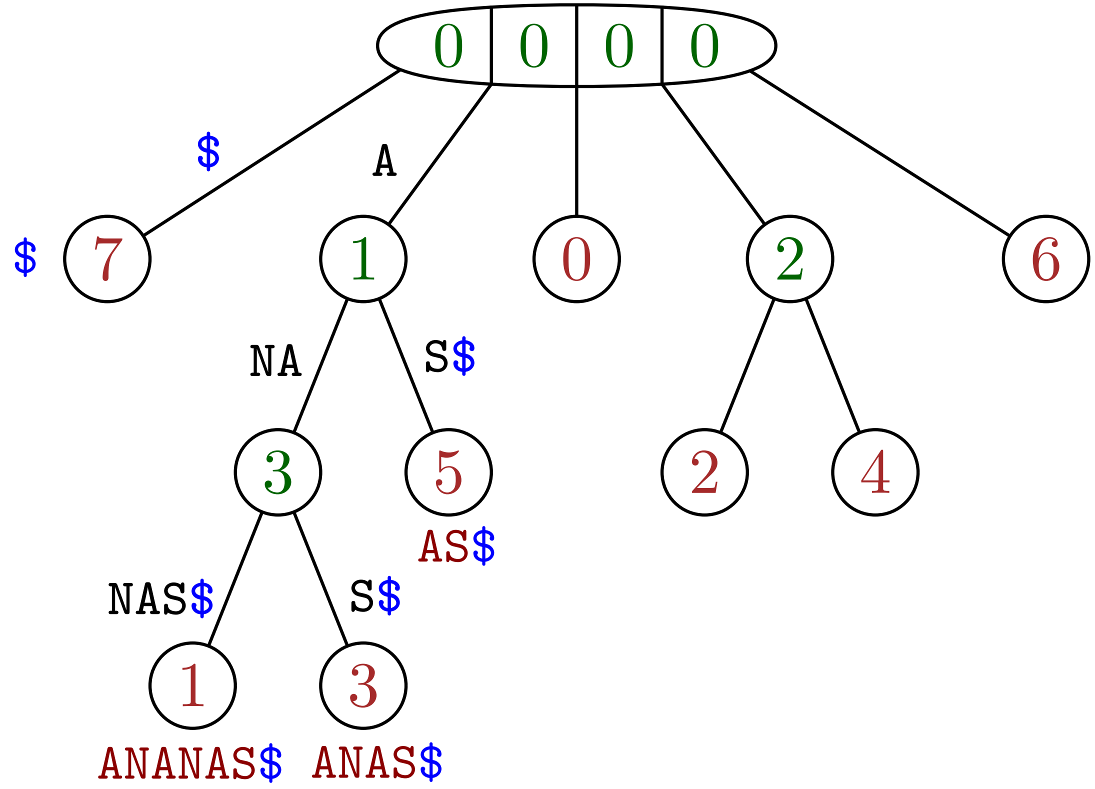
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

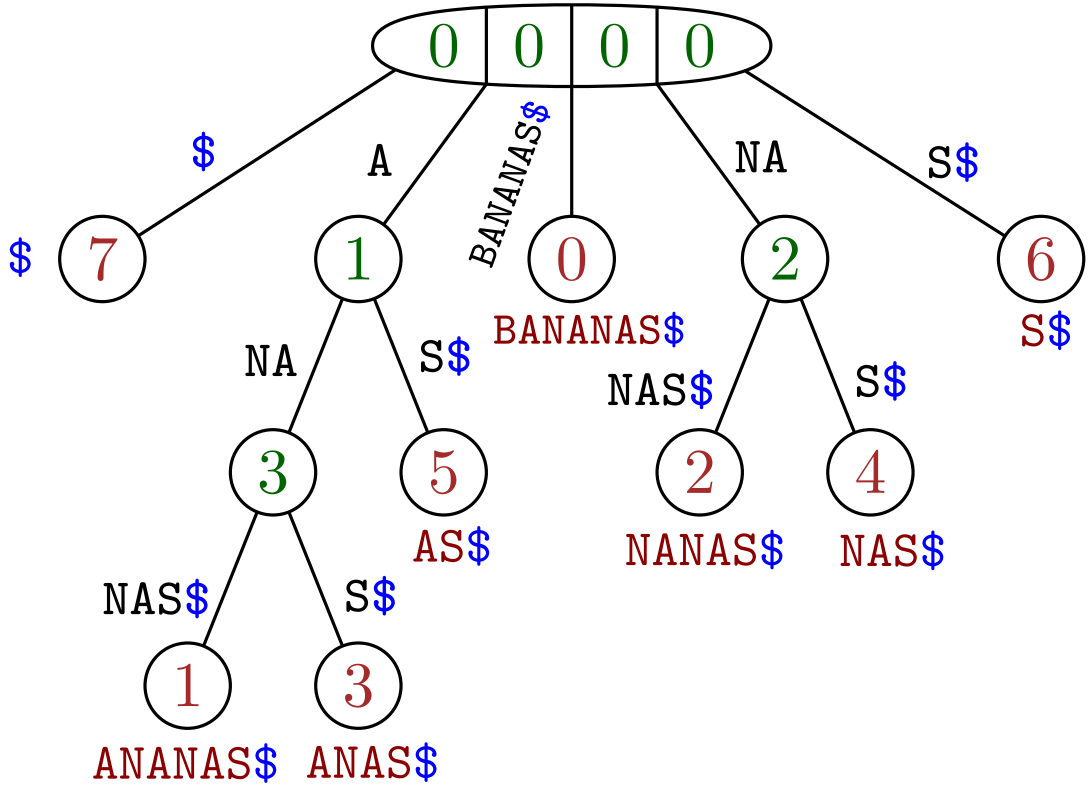
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

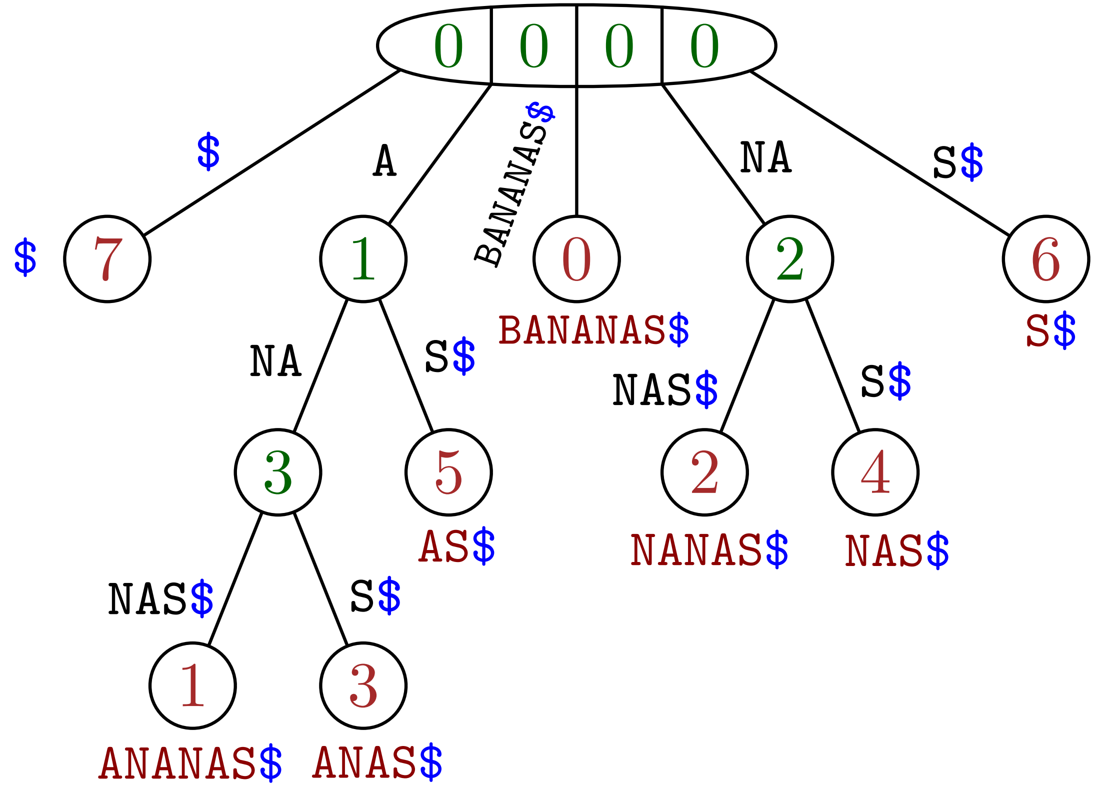
Edge labels are easy to reconstruct with a post-order visit



Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

Edge labels are easy to reconstruct with a post-order visit

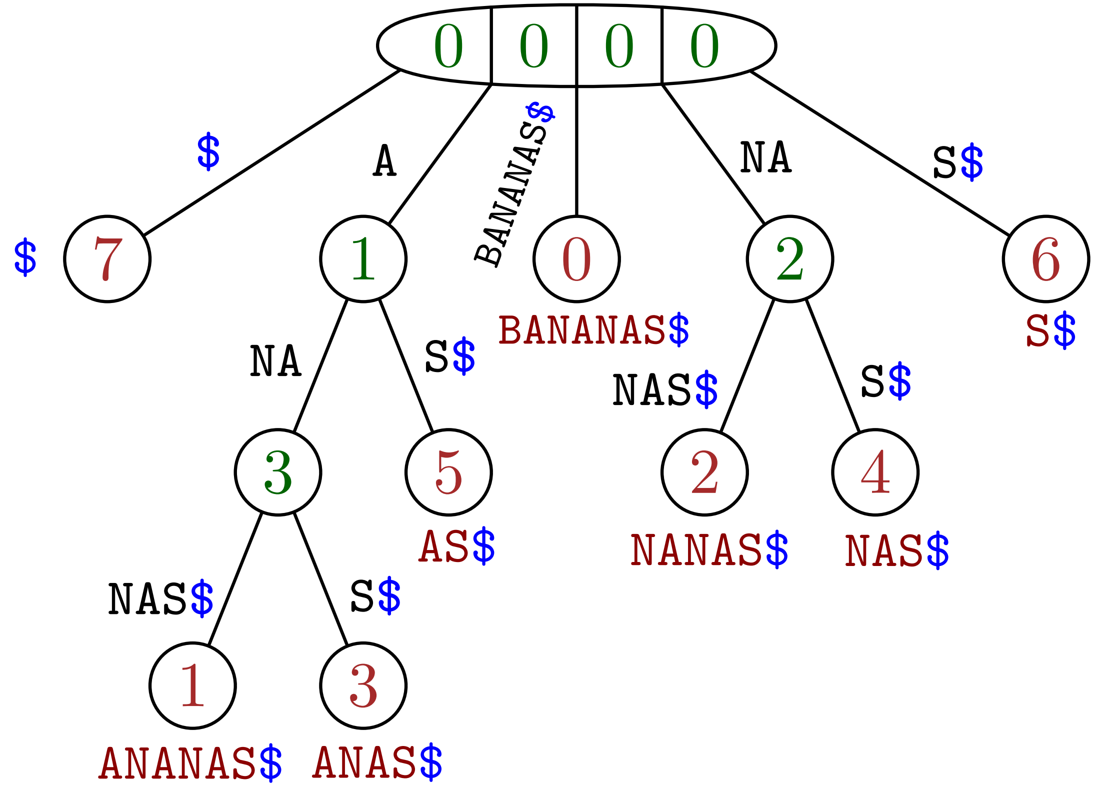


Construction time
(from Suffix + LCP Arrays):
$O(|T|)$

Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

Edge labels are easy to reconstruct with a post-order visit



Construction time (from Suffix + LCP Arrays): $O(|T|)$

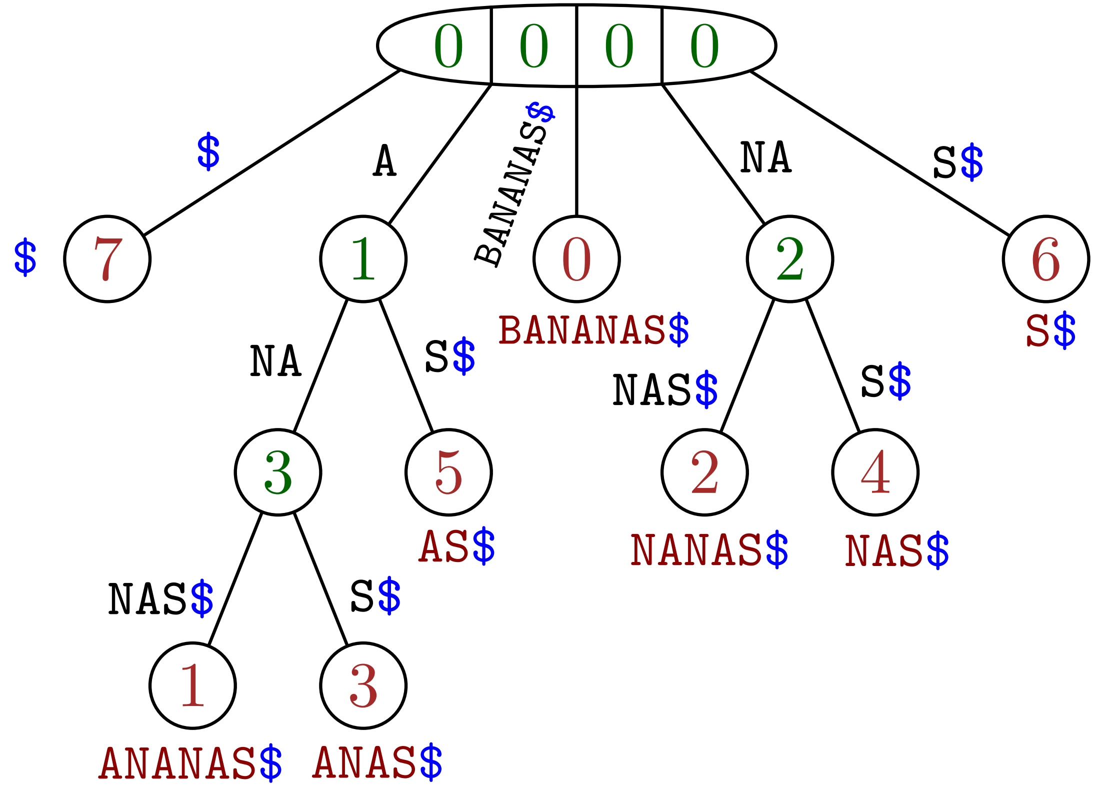
Suffix + LCP Arrays can be built in $O(|T|)$ time

[J. Kärkkäinen, P. Sanders, ICALP'03]

Suffix Arrays & Suffix Trees

Branching vertices are labelled with their *letter depth*, i.e., the number of letters of the prefix encoded in the path from the root to the vertex

Edge labels are easy to reconstruct with a post-order visit



Construction time
(from Suffix + LCP Arrays):
$O(|T|)$

Suffix + LCP Arrays can be built in $O(|T|)$ time

[J. Kärkkäinen, P. Sanders, ICALP'03]

$\Downarrow$

Suffix trees can be built in $O(|T|)$ time!